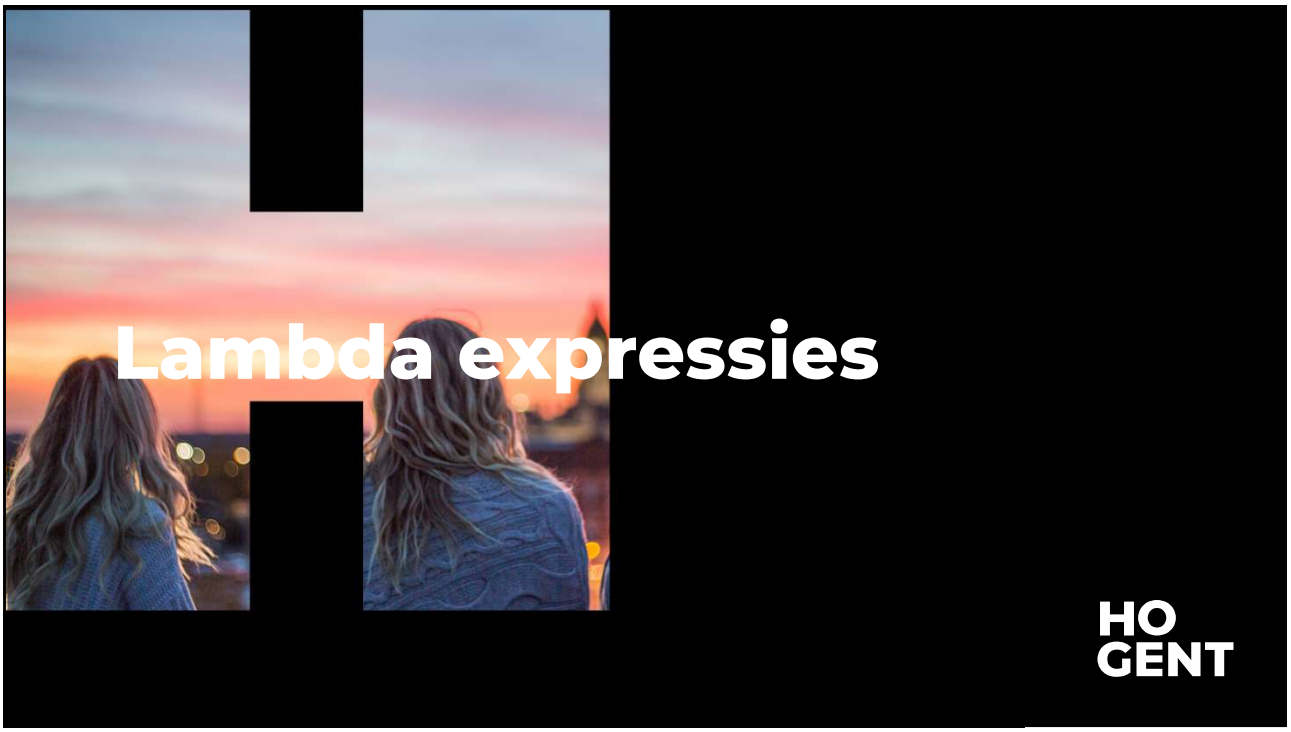

Inhoud

HO
GENT

1. Doelstellingen

Je bent in staat om volgende zaken te herkennen, toe te lichten, te definiëren, toe te passen en te implementeren:

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

HO GENT

2. Inleiding

- Een **functionele interface** is interface met exact één abstracte methode.
- Een **lambda expressie** implementeert de enige abstracte methode van een functionele interface, dwz dat de lambda expressie de functionele interface implementeert.
- Een lambda expressie creëert ook een instantie van een functionele interface.
- Lambda expressies zijn een stap in de richting van **functioneel programmeren**.

HO GENT

3. Implementatie van een interface

- Een interface kan je implementeren in een gewone klasse.
- Maar een interface kan je ook implementeren in:
  - een lokale klasse binnen een code blok of
  - in een anonieme klasse waar je in één stap een klasse declareert en instantieert

HO GENT

3. Implementatie van een interface

Voorbeeld:

```
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             public void greetSomeone(String someone) {
12                 System.out.println("Hello " + someone);
13             }
14         }
15
16         HelloWorld englishGreeting = new EnglishGreeting(); ③
17
```

① Declaratie van de interface HelloWorld

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

③ Instantiëren van een EnglishGreeting object.

HO GENT

3. Implementatie van een interface

Voorbeeld (vervolg):

```
18          // Anonymous class
19          HelloWorld frenchGreeting = new HelloWorld() { ④
20              public void greetSomeone(String someone) {
21                  System.out.println("Salut " + someone);
22              }
23          };
24
25          englishGreeting.greetSomeone("world"); ⑤
26          frenchGreeting.greetSomeone("Fred");
27      }
28
29      public static void main(String[] args) {
30          HelloWorldLocalClass myApp =
31              new HelloWorldLocalClass();
32          myApp.sayHello();
33      }
34  }
```

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

⑤ Gebruik van de objecten.

HO GENT

4. Anonieme klasse

- De definitie van een anonieme klasse gebeurt in een expressie die deel moet uitmaken van een statement.
- De syntax van de anonieme klasse expressie is analoog aan het aanroepen van een constructor en bestaat uit:
  - De new operator
  - De naam van de te implementeren interface
  - De klasse declaratie met daarin de implementatie van de methode

HO GENT

5. Lambda expressies

- Met een lambda expressie kan je een functionele interface implementeren en instantiëren in een expressie.
- Een lambda expressie wordt vaak gebruikt om een functionele interface te implementeren en te instantiëren en die instantie door te geven als parameter aan een methode.

HO GENT

5. Lambda expressies: voorbeeld

```
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             String name = someone;
12             System.out.println("Hello " + name);
13         };
14
15         dutchGreeting.greetSomeone("Pete");
16     }
17
18     public static void main(String[] args) {
19         HelloWorldLambdaExpression myApp =
20             new HelloWorldLambdaExpression();
21         myApp.sayHello();
22     }
23 }
```

① Lambda expressies kunnen enkel gebruikt worden om functionele interfaces te implementeren.

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

HO GENT

5.1. Syntax van Lambda expressies

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

- Parameterlijst:
  - De Java compiler kan vaak uit de context het type van de parameters afleiden. Je kan dit type dus meestal weglaten bij de parameters.
  - Als er maar één parameter is mag je ook de haakjes weglaten.
- De statements bestaan uit één expressie of uit een blok statements.
  - Bij één enkele expressie wordt deze expressie geëvalueerd en het resultaat teruggeven.

HO GENT

5.1. Syntax van Lambda expressies

- Voorbeeld:

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

is equivalent aan:

```
10          HelloWorld dutchGreeting = (String someone) -> { ②
11              String name = someone;
12              System.out.println("Hello " + name);
13          };
```

HO GENT

6. Methode referenties

- Lambda expressies zijn enkel van toepassing op functionele interfaces.
- In sommige gevallen roept de lambda expressie enkel een bestaande methode aan.
- In dit geval gebruiken we een methode referentie.

HO GENT

6. Methode referenties

Voorbeeld:

```
 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("Hello " + name);
15    }
```

① Een lambda expressie die gebruik maakt van een methode referentie

② De methode waarnaar gerefereerd wordt.

HO GENT

6.1. Soorten methode referenties: 4 soorten

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

HO GENT

7. Functionele interfaces

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

HO GENT

8. Basis functionele interfaces: uit de API

Functionele interface	Prototype - abstracte methode	Beschrijving
Function<T, R> BiFunction<T,U,R>	R apply(T t) R apply(T t, U u)	Het return type is verschillend van het type van de argumenten.
Predicate<T> BiPredicate<T,U>	boolean test(T t) boolean test(T t, U u)	Neemt een of twee argumenten en geeft een boolean terug.
Consumer<T> BiConsumer<T,U>	void accept(T t) void accept(T t, U u)	Neemt een of twee argumenten, het return type is void.
Supplier<T>	T get()	Neemt geen argumenten en geeft een waarde terug.
BinaryOperator<T>	T apply(T t1, T t2)	Beide argument types en het return type zijn identiek.
UnaryOperator<T>	T apply(T t)	Het argument type en return type zijn identiek.

8. Basis functionele interfaces: uit de API

- 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

HO GENT

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Function.html

8.1. Function

```
@FunctionalInterface
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is `apply(Object)`.

Since:
1.8

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
`default <V> Function<T, V>`	`andThen(Function<? super R,? extends V> after)`	Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.
`R`	`apply(T t)`	Applies this function to the given argument.
`default <V> Function<V, R>`	`compose(Function<? super V,? extends T> before)`	Returns a composed function that first applies the `before` function to its input, and then applies this function to the result.
`static <T> Function<T, T>`	`identity()`	Returns a function that always returns its input argument.

8.1. Function: voorbeeld

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

HO GENT

8.1. Function: voorbeeld

```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 geeft het binnenkomende argument terug.

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

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Predicate.html

8.2. Predicate

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

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since:
1.8

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
default Predicate<T>	and(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> Predicate<T>	isEqual(Object targetRef)	Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).
default Predicate<T>	negate()	Returns a predicate that represents the logical negation of this predicate.
static <T> Predicate<T>	not(Predicate<? super T> target)	Returns a predicate that is the negation of the supplied predicate.
default Predicate<T>	or(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	test(T t)	Evaluates this predicate on the given argument.

8.2. Predicate: voorbeeld

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

HO GENT

8.2. Predicate: voorbeeld

```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> lessThen18 = i -> (i < 18);  ①

        // Test de voorwaarde
        System.out.println(lessThen18.test(10));

        // Predicate met voorwaarde "> 12"
        Predicate<Integer> greaterThen12 = (i) -> i > 5;

        // Test de voorwaarde
        System.out.println(greaterThen12.test(10));

```

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

HO GENT

8.2. Predicate: voorbeeld

```
26          // Samengestelde predicate
27          Predicate<Integer> lessThen18AndGreaterThen12 = lessThen18.and
(greaterThen12);
28
29          boolean result = lessThen18AndGreaterThen12.test(16); ②
30          System.out.println(result);
31
32          // Negatie van een Predicate
33          boolean result2 = lessThen18AndGreaterThen12.negate().test(16); ③
34          System.out.println(result2);
35
36          //passing Predicate into function
37          pred(10, (i) -> i > 7);
38
39          // OR Predicate
40          Predicate<String> hasLengthOf10 = t -> t.length() > 10;
41          Predicate<String> containsLetterA = p -> p.contains("A");
42          String containsA = "And";
43
44          boolean outcome = hasLengthOf10.or(containsLetterA).test(containsA); ④
45          System.out.println(outcome);
46      }
47 }
```

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.

HO GENT

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Consumer.html

8.3. Consumer

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

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, `Consumer` is expected to operate via side-effects.

This is a functional interface whose functional method is `accept(Object)`.

Since:
1.8

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
void	accept(T t)	Performs this operation on the given argument.
default Consumer<T>	andThen(Consumer<? super T> after)	Returns a composed Consumer that performs, in sequence, this operation followed by the after operation.

HO GENT

8.3. Consumer: voorbeeld

```java
public interface Consumer<T> {
    void accept(T t);
}
```

HO GENT

8.3. Consumer: voorbeeld

```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);

```

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

HO GENT

8.3. Consumer: voorbeeld

```
15        // Consumer to display a number
16        Consumer<List<Integer>> displayList = a -> System.out.println(a);
17
18        // Implement display using accept()
19        displayList.accept(list);
20
21        // Consumer die elk element in een lijst verdubbelt
22        Consumer<List<Integer>> addTen = a -> {
23            for (int i = 0; i < list.size(); i++)
24                list.set(i, 2 * list.get(i));
25        };
26
27        // Samengestelde Consumer: verdubbel elk element in de lijst en
28        // druk elk element af op het scherm
29        addTen.andThen(displayList).accept(list); ②
30    }
31 }
```

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

HO GENT

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Supplier.html

8.4. Supplier

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

Represents a supplier of results.

There is no requirement that a new or distinct result be returned each time the supplier is invoked.

This is a functional interface whose functional method is `get()`.

Since:
1.8

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
T	`get()`	Gets a result.

8.4. Supplier: voorbeeld

```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();

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

HO GENT

8.5. BinaryOperator en UnaryOperator

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

HO GENT