Todo: paginatitels aanpassen zodat structuur duidelijker wordt
Todo: slides verbergen/schrappen om een vlottere flow in de les te verkrijgen

Inhoud

Nested class
Inner class
Local inner class
Anonymous class
Lambda
Functional Interface
Method Reference

There are **four kinds of nested class in Java**. In brief, they are:
static class: declared as a static member of another class
inner class: declared as an instance member of another class
local inner class: declared inside an instance method of another class
anonymous inner class: like a local inner class, but written as an expression which returns a one-off object

Voorkennis

Klassen

Interfaces

static

1 klasse, 1 bestand

new ActionListener() { … }

ActionListener is gezien bij GUI applicaties vorig jaar, maar is een Interface.

Static nested class

```
public class OuterClass {
    public static class NestedClass {
        ....
    }
}
```

Gebruik (buitenaf):

```
OuterClass.NestedClass nested = new OuterClass.NestedClass();
```

NestedClass noemt men een static nested class. Static: de klasse hoort in de outer klasse, niet in een object van de outer klasse. De klasse kan wel geinstantieerd worden.

Static nested class

```
public class OuterClass {
    private static int classField = 1;

    public static class NestedClass {
        private int nestedField;
        public NestedClass() {
            nestedField = classField++;
        }
    }
}
```

NestedClass heeft toegang tot private classField.

Static nested class

```
public class OuterClass {
    private int field = 1;

    public static class NestedClass {
        public void setField(OuterClass outer, int val) {
            outer.field = val;
        }
    }
}
```

NestedClass heeft onrechtstreeks toegang tot de private data van instanties van OuterClass.
Merk op, setField kan hier evengoed static zijn. NestedClass (object) heeft geen rechtstreekse toegang tot field, want het heeft geen standaard instantie van OuterClass object om mee te werken.

Static nested class

```
public class OuterClass {
    private NestedClass nested;

    public OuterClass() {
        nested = new NestedClass();
    }

    public int getValue() {
        return nested.calculate();
    }

    private static class NestedClass {
        private int calculate() { … };
    }
}
```

NestedClass kan private, package, protected of public zijn. Bij private kan alleen OuterClass er aan, bijv. in de constructor. Bij protected enkel subklassen _van OuterClass_ en klassen in hetzelfde package.
Verder kan de OuterClass ook aan de private methodes en velden van de NestedClass

Static nested class: voorbeeld

```
public class ColorEnums {
    public enum Color {
        RED, GREEN, BLUE, YELLOW, …;
    }
}
```

Gebruik:

```
ColorEnums.Color color = ColorEnum.Color.RED;
```

Bij enum mag static worden weggelaten.

Nested class

Samenvatting:

- Static nested class kan geïnstantieerd worden
- Static nested class (instantie) zit in de Outer class
- private ≈ in hetzelfde bestand

9

Verder zijn extends, abstract, final, enz. zijn ook mogelijk op nested classes. “In hetzelfde bestand” is een goede techniek om het te onthouden, mits rekening te houden met static. NestedClass1 kan bijvoorbeeld aan NestedClass2.privateData.

Inner class

```
public class OuterClass {
    private int field = 1;

    public class InnerClass {
        private int nestedField;
        public InnerClass() {
            nestedField = field;
        }
    }
}
```

Inner class is niet static en zit conceptueel in een object van OuterClass i.p.v. in de class zelf. Het directe gevolg is rechtstreekse toegang tot private non-static velden en methodes.
Cursus over inner class: “De geneste klasse kan alleen gebruikt worden binnen de context van de nestende klasse.” De naam van het klassebestand wordt OuterClass $InnerClass.class

Inner class

```
public class OuterClass {
    public void doSomething() {
        InnerClass inner = new InnerClass();
        inner.doMethod();
    }

    public class InnerClass {
        public void doMethod () { … }
    }
}
```

Aanmaak van een object van de inner class binnen de nestende klasse.

Inner class

```
public class OuterClass {
    public void doSomething() {
        InnerClass inner = this.new InnerClass();
        inner.doMethod();
    }

    public class InnerClass {
        public void doMethod () { … }
    }
}
```

Exact hetzelfde, nu met expliciete this-verwijzing. De InnerClass is in de context van een object van OuterClass en vereist dus een referentie. In de vorige slide was het gewoon impliciet.

Inner class

```
public class OuterClass {
    public class InnerClass {  }
}

OuterClass outer = new OuterClass();
OuterClass.InnerClass inner;
inner = outer.new InnerClass();
```

Hetzelfde, van buitenaf. Het datatype is dus OuterClass.InnerClass (volgens cursus. Na import mijnpkg.OuterClass.InnerClass kan InnerClass op zich ook)

Inner class

```
public class OuterClass {
    private int field = 1;

    public class InnerClass {
        private int field;
        public InnerClass(int field) {
            this.field = field;
            OuterClass.this.field = field;
        }
    }
}
```

Shadowing “field”. Met OuterClass.this kom je aan de context, de instantie van OuterClass

Local inner class

```
public class OuterClass {
    public void method() {
        final int CONST = 5;
        int val = 7;

        class LocalInnerClass {
            ...
            System.out.println(CONST); // OK
            System.out.println(val);       // OK (val wijzigt niet).
            ...
        }
        LocalInnerClass local = new LocalInnerClass();
    }
}
```

Een local inner class is een klasse gedefinieerd in een methode. Enkel binnen de methode kan de klasse geinstantieerd worden. De local inner class kan aan lokale variabelen die niet veranderen. De compiler beschouwt ze als final (sinds Java 8).

Local inner class

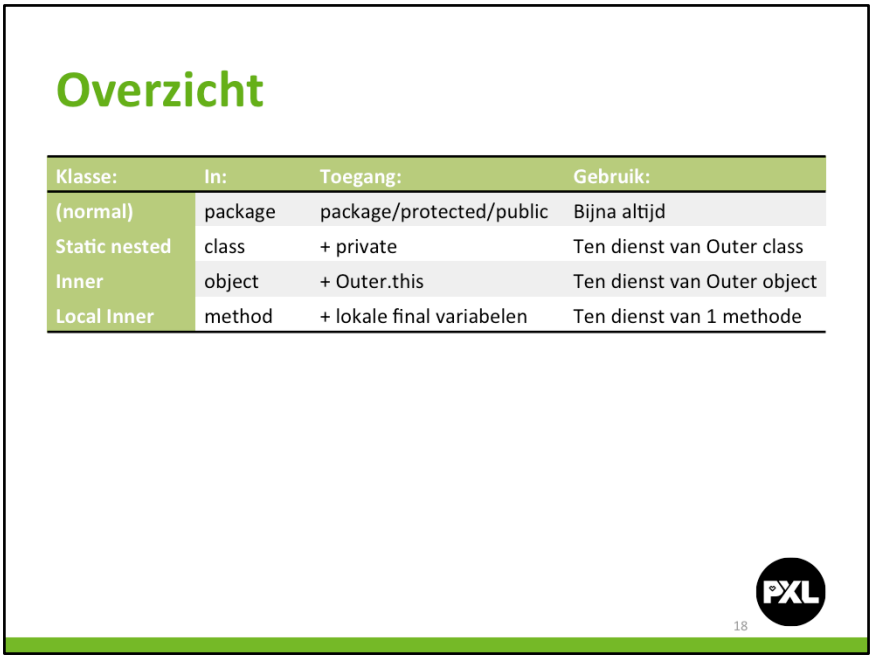
```
public class OuterClass {
	public Object getInner() {
		int val = 7;
		class LocalInnerClass {
			public String toString() {
				val++; // Fout
				return "Inner " + val;
			}
		}
		return new LocalInnerClass();
	}
}

public static void main(String[] args) {
	OuterClass outer = new OuterClass();
	outer.getInner().toString();
}
```

Opgelet: als de lokale variabelen wijzigen en niet onveranderlijk zijn, kan de local inner class niet aan hun waarden. Na het beëindigen van de methode, bestaan de lokale variabelen niet meer, maar het object eventueel nog wel.

Local inner class

```
public class OuterClass {
    private int val = 1;
    public void method() {
        class LocalInnerClass {
            …
            System.out.println(OuterClass.this.val); // OK
            …
        }
        LocalInnerClass local = new LocalInnerClass();
    }
}
```

Een local inner class is een inner klasse en kan dus wel aan de instantievariabelen van haar bijhorende object. Daarnaast, een local inner class kan net als lokale variabelen niet private, public of protected zijn.



Korte samenvatting. In geeft de context aan, in welke ‘parent container’ het conceptueel thuishoort. Toegang is waar de klasse kan aankomen. Het verschil tussen static nested en inner is niet de private instantie velden, het is dat de inner klasse altijd een Outer instantie heeft; een static nested kan aan de private velden als het Outer instantie krijgt om mee te werken. Gebruik geeft een zeer beknopte uitleg wanneer de klasse te gebruiken. Static nested kan nuttig zijn als er een zeer sterk verband is tussen de Outer en de nested. Bij het schrappen van de Outer class is de nested class waardeloos geworden. Bij Inner gaat het verder: het schrappen van 1 Outer object, maakt het inner object waardeloos. Bij local inner wordt na het schrappen van die ene methode de local class waardeloos. (vage leidraad, korrel zout, bron: oracle.com)

Anonymous class

```
public class OuterClass {
    public void method() {
        class SubClass extends SuperClass {
            // Vervangen methodes
        }
        SuperClass object1 = new SubClass();

        class InterfaceClass implements Interface {
            // Implementatie van methodes
        }
        Interface object2 = new InterfaceClass();
    }
}
```

object1 is een object van een subklasse van de klasse SuperClass. In deze subklasse worden alleen methodes van de superklasse vervangen. object2 is een object van een klasse die de interface Interface implementeert. We kunnen dit ook m.b.v. anonieme klassen realiseren.

Anonymous class

```
public class OuterClass {
    public void method() {
        SuperClass object1 = new SuperClass() {
            // Vervangen methodes
        };


        Interface object2 = new Interface() {
            // Implementatie van methodes
        };

    }
}
```

Opgelet, vermits de klasse anoniem is, kan je geen constructor definiëren en kunnen nieuwe methodes niet worden opgeroepen van buitenaf (want niet gedefinieerd in SuperClass of Interface).

Toepassing: iterator

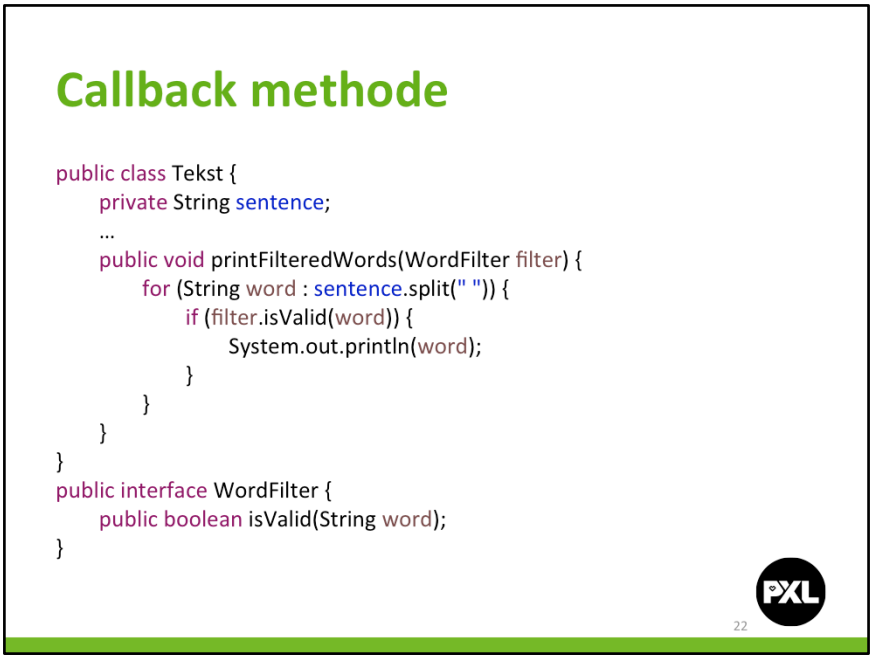
```
Iterator it;
it = collection.iterator();
while (it.hasnext()) {
    Object o = it.next();
    // use o …
}
```

```
for (Object o : collection) {
    // use o …
}
```

De Iterator interface kan geïmplementeerd worden in een nested class in de collection klasse.

21

Iterable collection; Iterable is de interface die de methode iterator aanbiedt die een object van de interface Iterator aanbiedt. (toepassing uit cursus p. 30)



Deze klasse laat toe om woorden af te drukken die aan een bepaald criterium voldoen. De aanroeper kiest het criterium door een object mee te geven dat de interface WordFilter implementeert. De methode in het object meegegeven door de aanroeper wordt telkens opgeroepen om elk woord te testen. Zo een methode noemen we een callback-methode.
Niet getoond: de constructor waarmee sentence wordt ingesteld.

Callback methode

```
public class TekstApp {
    ...
    Text text = new Text("Hello this is an example sentence");
    text.printFilteredWords(new WordFilter() {
        @Override
        public boolean isValid(String word) {
            return word.contains("e");
        }
    });
    ...
}
```

23

Voorbeeld van het gebruik van WordFilter en de implementatie van een callback-methode. Merk op, bij het debuggen springt de code per woord in deze methode om te testen of word een e bevat.

Callback methode

```
public class TekstApp {
    ...
    Text text = new Text("Hello this is an example sentence");
    text.printFilteredWords(new WordFilter() {
        @Override
        public boolean isValid(String word) {
            return word.length() > 4;
        }
    });
    ...
}
```

24

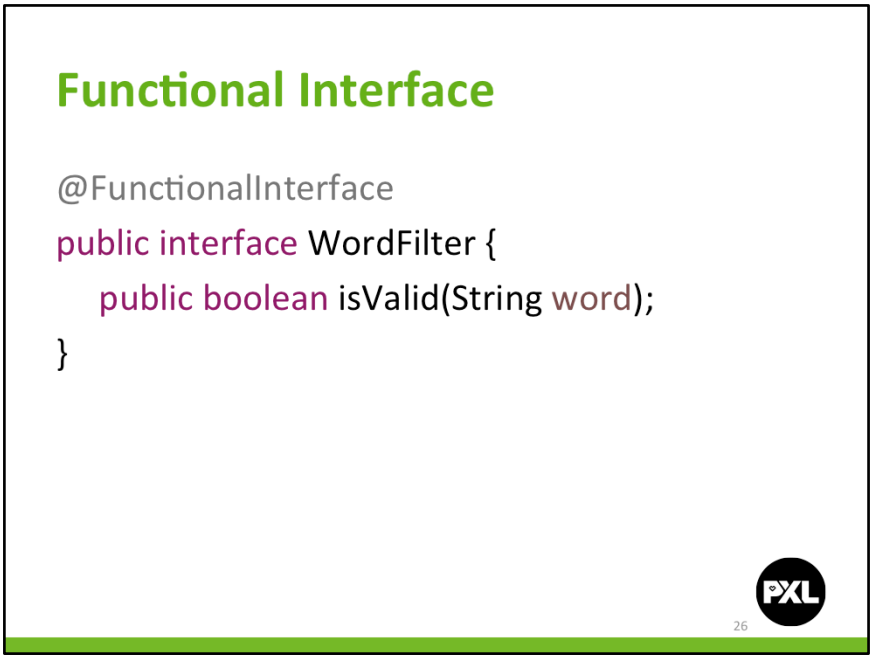
Andere filter. Hier testen we of de lengte groter is dan 4. Merk op, dit hadden we ook direct in printFilteredWords kunnen zetten, maar nu is de conditie los van de klasse Tekst.

Callback methode

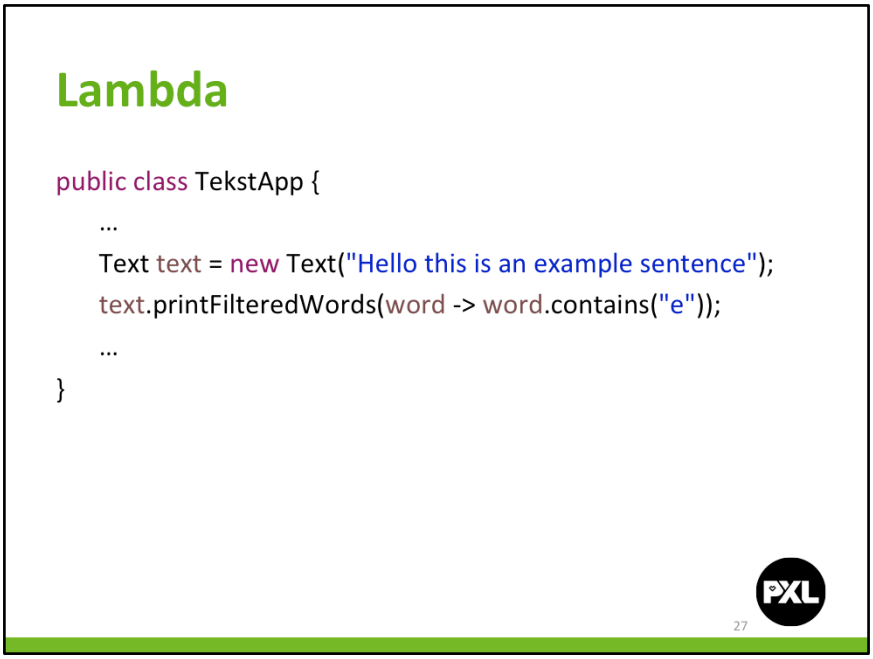
```
public class TekstApp {
    ...
    Text text = new Text("Hello this is an example sentence");
    text.printFilteredWords(new WordFilter() {
        @Override
        public boolean isValid(String word) {
            return word.startsWith("a");
        }
    });
    ...
}
```

25

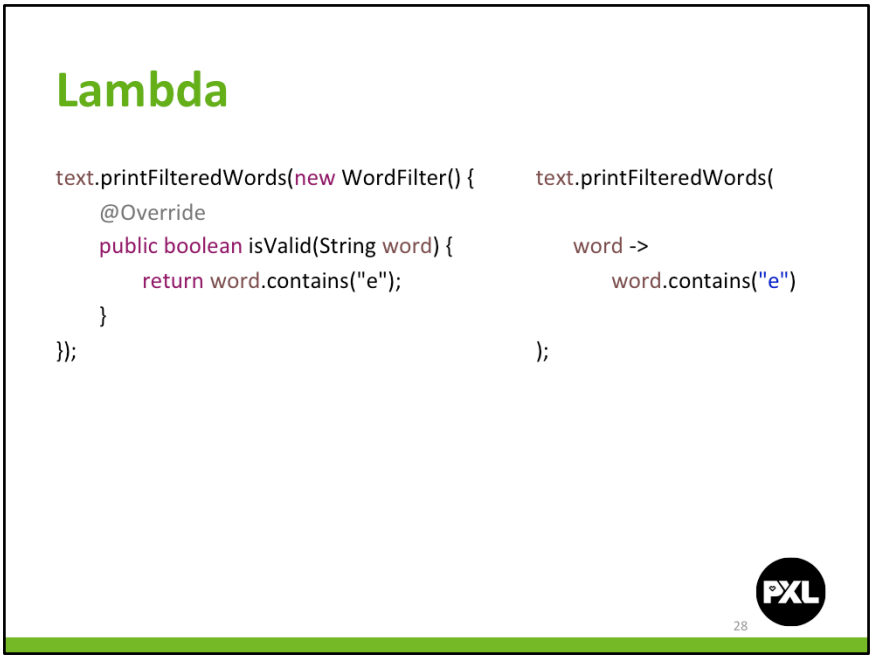
Andere filter. Dit toont de flexibiliteit. We hebben succesvol het filter-criterium losgetrokken van de iteratie en het outputten. In conceptuele termen hebben we de conditie variabel gemaakt; de WordFilter is een variabele die niet het resultaat van een conditie is (een boolean), maar de bepaling ervan, de conditie zelf. Enige nadeel: nogal omslachtige code. De essentie is de code in isValid.



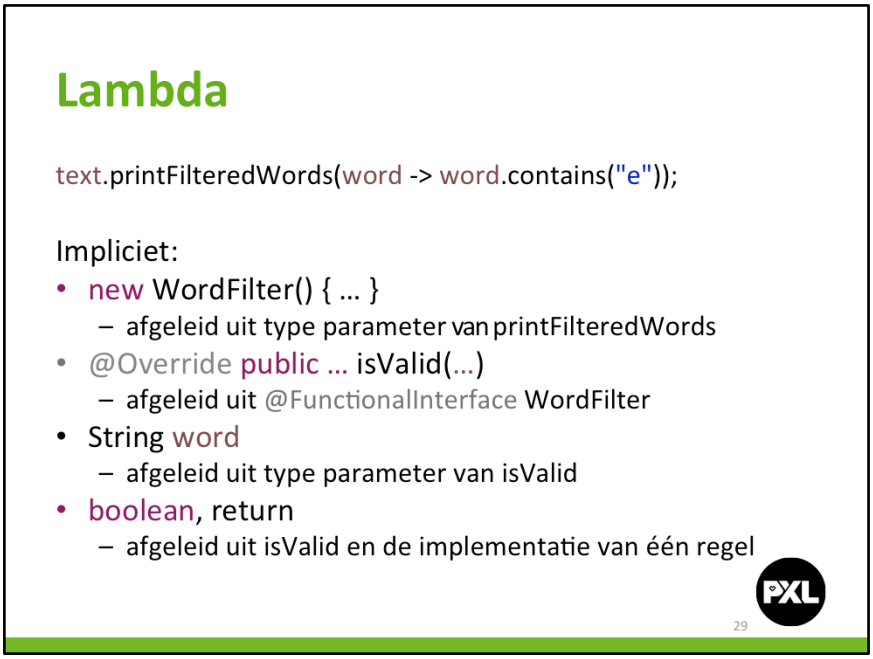
Zo'n interface met slechts één methode, noemen we een functionele interface. De objecten en klassen die deze interface implementeren hebben namelijk enkel een eenvoudige 'functie', ze gebruiken geen extra eigenschappen van het object. In feite is dus geen object met eigen data nodig, enkel een stukje code dat uitgevoerd moet worden, een functie. We kunnen expliciet opleggen door de annotatie FunctionalInterface toe te voegen.



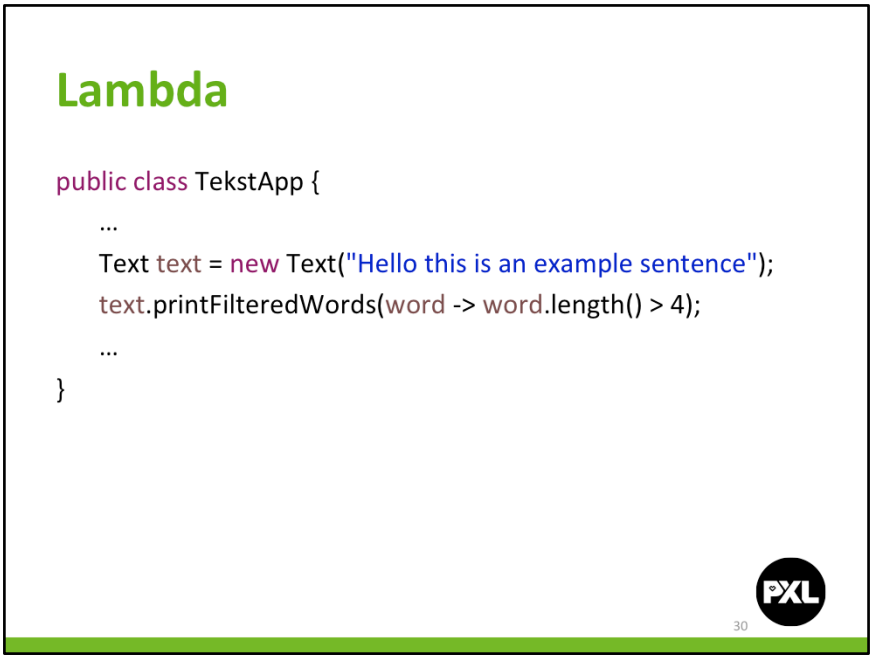
Weg met de omslachtige code. De essentie, de implementatie van isValid is behouden. Het enige wat nog overblijft van de definitie is de naam van de parameter. (volgende slide vergelijking)



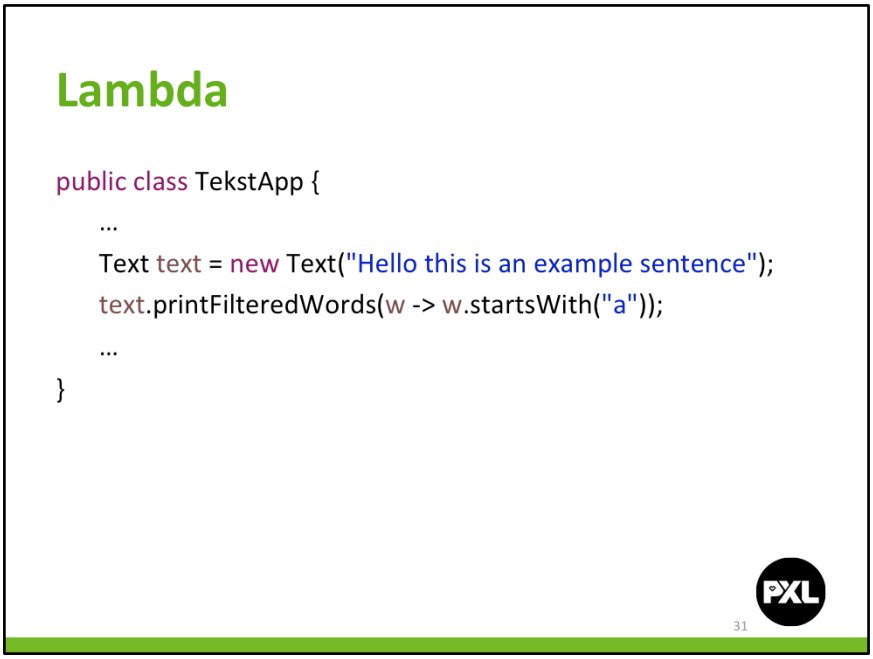
Vergelijking oud/nieuw. Ze doen hetzelfde. Enkel behouden is de naam van de parameter (zodat die niet 'word' hoeft te zijn) en de code. Al de rest is niet meer nodig. Het rechtse is wel degelijk een 'methode' waar de debugger in komt om de conditie te testen. Meer uitleg over wat weg is op de volgende slide.



Herinnering: FunctionalInterface had slechts één methode, dus het kan enkel die zijn die wordt geïmplementeerd. Doordat word.contains("e") de enige implementatie is én isValid een boolean vereist, is return overbodig: enkel die ene regel kan een boolean teruggeven.



Weg met de omslachtige code. De essentie, de implementatie van isValid is behouden. Het enige wat nog overblijft van de definitie is de naam van de parameter. (volgende slide vergelijking)



De naam van de parameter mag wijzigen.

Lambda

Syntax:

- WordFilter filter = word -> word.charAt(2) == 'i';
- WordFilter filter = word -> {
  return word.charAt(2) == 'i';
  };
- WordFilter filter = (word) -> word.charAt(2) == 'i';
- WordFilter filter = (String word) -> word.charAt(2) == 'i';

32

Langere syntax, exact hetzelfde (hier). Langere syntax laat andere mogelijkheden (volgende slides). In block {…} is return wel nodig. Daarnaast, hier leidt Java af dat het WordFilter wordt uit de assignment, i.p.v. parameter tot nog toe.

Lambda

Voorbeeld:

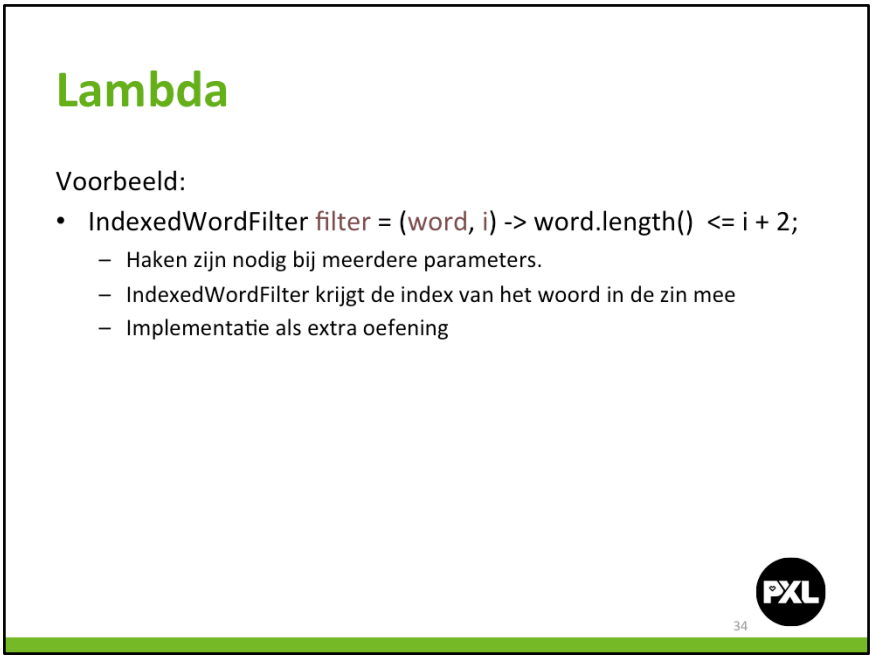
- WordFilter filter = word -> {

```
try {
    int value = Integer.parseInt(word);
    return value > 10;
} catch (NumberFormatException nfe) {
    return false;
}
};
```

  - Accolades en return zijn nodig bij meerdere statements.

33

Voorbeelden met langere syntax.



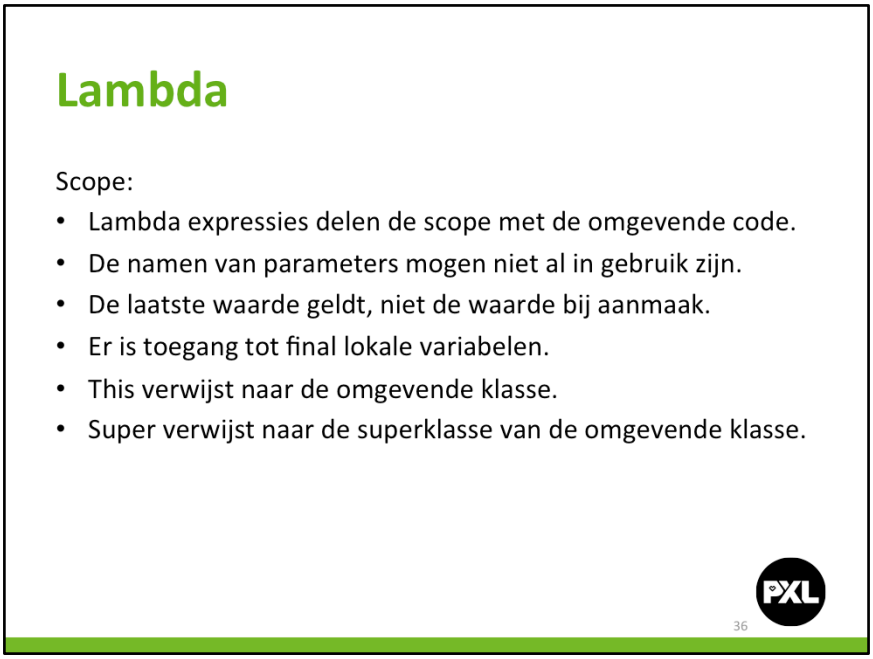
Voorbeelden met langere syntax; Voorbeeld tekst hier: “In het zware boek van gelukkige tante Francine” wordt “In het boek van tante Francine”. Datatype (String word, int i) is nodig als de compiler het niet uit de context kan afleiden (cursus). Vermits er echter altijd een Functionele interface moet zijn, is er zo goed als altijd een context. Uitzondering: als generic functionele interfaces worden gebruikt in een situatie waar een generische parameter niet kan bepaald worden. Uitzonderlijk (details: zie http://stackoverflow.com/questions/31370113/when-does-java-require-explicit-type-parameters).

Lambda

```
public class TekstApp {
    private String start;
    ...
    Text text = new Text("bob blijft achter de blauwe auto");
    start = "a";
    WordFilter filter = w -> w.startsWith(start);
    start = "b";
    text.printFilteredWords(filter);
    ...
}
```

35

Lambdas delen een scope met de omgevende code, ze hebben niet hun eigen scope (uitleg op volgende slide).



Zoals bij local inner classes, lokale variabelen die niet wijzigen zijn final.

Method reference

```
@FunctionalInterface
public interface WordProcessor {
    public String process(String word);
}

// in Text.java
public void printProcessedWords(WordProcessor processor) {
    for (String word : sentence.split(" ")) {
        System.out.println(processor.process(word));
    }
}
```

37

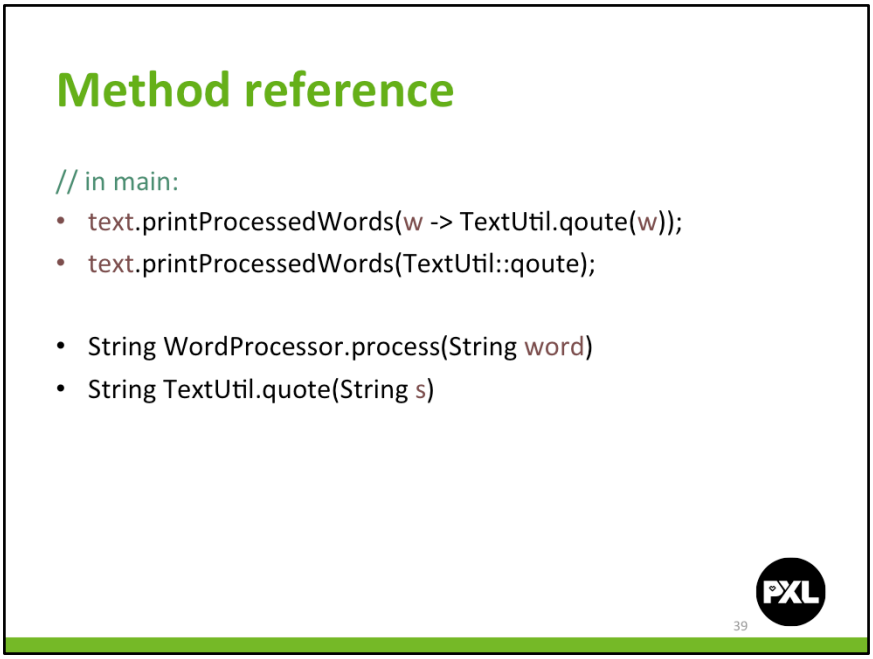
Deze interface geeft String terug i.p.v. boolean: het verwerkt de woorden/input en geeft tekst/output terug.

Method reference

```
public interface TextUtil {
    public static String quote(String str) {
        return String.format("<<%s>>", str);
    }
}

// in main:
text.printProcessedWords(w -> TextUtil.qoute(w));
```

38

We willen een bepaalde quoting techniek meerdere keren gebruiken. We besluiten de code onder te brengen in een utility-interface (cursus)



Deze twee doen juist hetzelfde. De methode quote van TextUtil heeft namelijk hetzelfde aantal en type parameters en hetzelfde return type als WordProcessor.process en kan daarom rechtstreeks ingezet worden in een lambda expression.

Method reference

Syntax:

qualifier::identifier

Mogelijkheden:

val -> ClassName.staticMethod(val)	ClassName::staticMethod
val -> myObject.method(val)	myObject::method
(Type val) -> val.method()	Type::method
val -> new ClassName(val)	ClassName::new

40

De verschillende vormen van method references. Type is hier expliciet, ook al is dat bij een gewone lambda niet nodig.

Method reference

data -> TextUtil.quote(data)	TextUtil::quote
woord -> vertaler.vertaal(woord)	vertaler::vertaal
tekst -> System.out.println(tekst)	System.out::println
(String w) -> w.toUpperCase()	String::toUpperCase
(naam, leeftijd) -> new Persoon(naam, leeftijd)	Persoon::new

41

Concrete voorbeelden. Merk op dat de parameter telkens verdwijnt bij het omzetten in een method reference

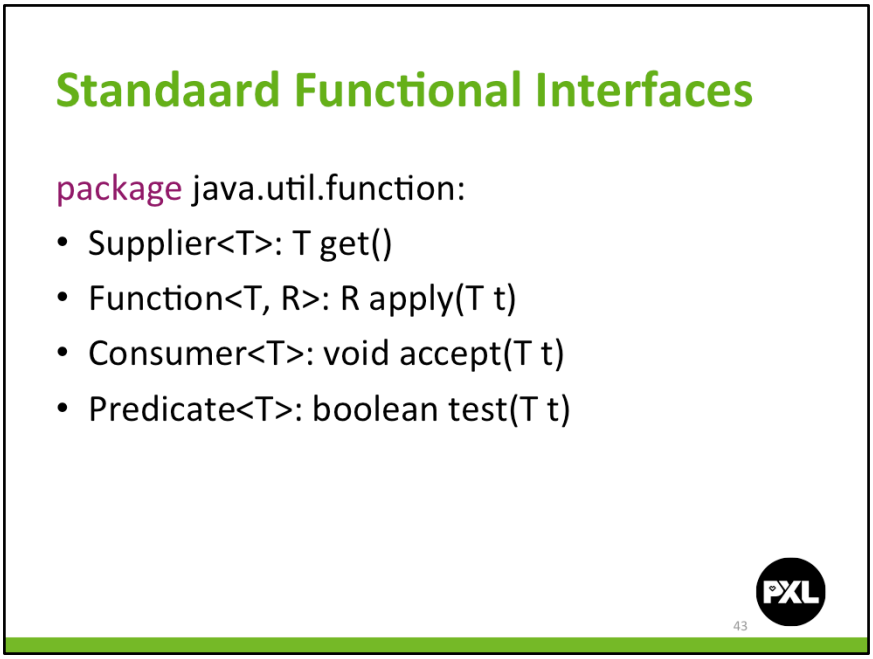
Method reference

Voorbeelden

data -> show(data)	this::show
getal -> bereken(getal, 5)	/
persoon -> super.toon(persoon)	super::toon
tekst -> tekst.indexOf(",")	/
tekst -> tekst.concat(tekst)	/

42

De verschillende vormen van method references. Details: p 36 tot 40.



Supplier levert items, Function berekent er een ander item mee en Consumer doet er iets mee. Dus krijg je vaak: Supplier > Function > Function > Consumer. Predicate is een conditie, typisch gebruikt bij filter.

Standaard Functional Interfaces

```
public class Text {
    private int i = 0;
    public Supplier<String> split(String teken) {
        i = 0;
        String[] data = text.split(teken);
        return () -> i < data.length ? data[i++] : null;
    }
    …
}
```

44

Voorbeeld van hoe een Supplier<String> te verkrijgen

Standaard Functional Interfaces

```
public class TestApp {
    public static void main(String[] args) {
        Text text = new Text("test 1 2 3 test");
        Supplier<String> woorden = text.split(" ");

        String woord = woorden.get();
        while (woord != null) {
            System.out.println(woord);
            woord = woorden.get();
        }
    }
}
```

45

Voorbeeld van hoe een Supplier<String> kan worden gebruikt. Dit soort code zou heel herkenbaar moeten zijn. Voordeel: het verkrijgen van de woorden staat los van het gebruik. En je kan vanaf het eerste woord dat je krijgt het beginnen verwerken.

Standaard Functional Interfaces

```
public class User {
    public Supplier<String> lees() {
        Scanner lezer = new Scanner(System.in);
        return () -> lezer.next();
    }
    ...
}
```

46

Je kan ook woorden vragen aan de gebruiker, over het netwerk, … Supplier<T> laat toe om het verkrijgen van T los te maken van het verwerken van T. De andere Standaard Functional Interfaces komen terug in de oefeningen en in het volgende hoofdstuk bij Streams.