

3 Reflection en Annotations

Programmeren 2 – Java

2017 - 2018

KdG Karel de Grote
Hogeschool

Kris Behiels
Jan De Rijke
Mark Goovaerts

Programmeren 2 - Java

1. Herhaling en Collections

2. Generics en documenteren

3. Annotations en Reflection

4. Testen en logging

5. Design patterns (deel 1)

6. Design patterns (deel 2)

7. Lambda's en streams

8. Persistentie (JDBC)

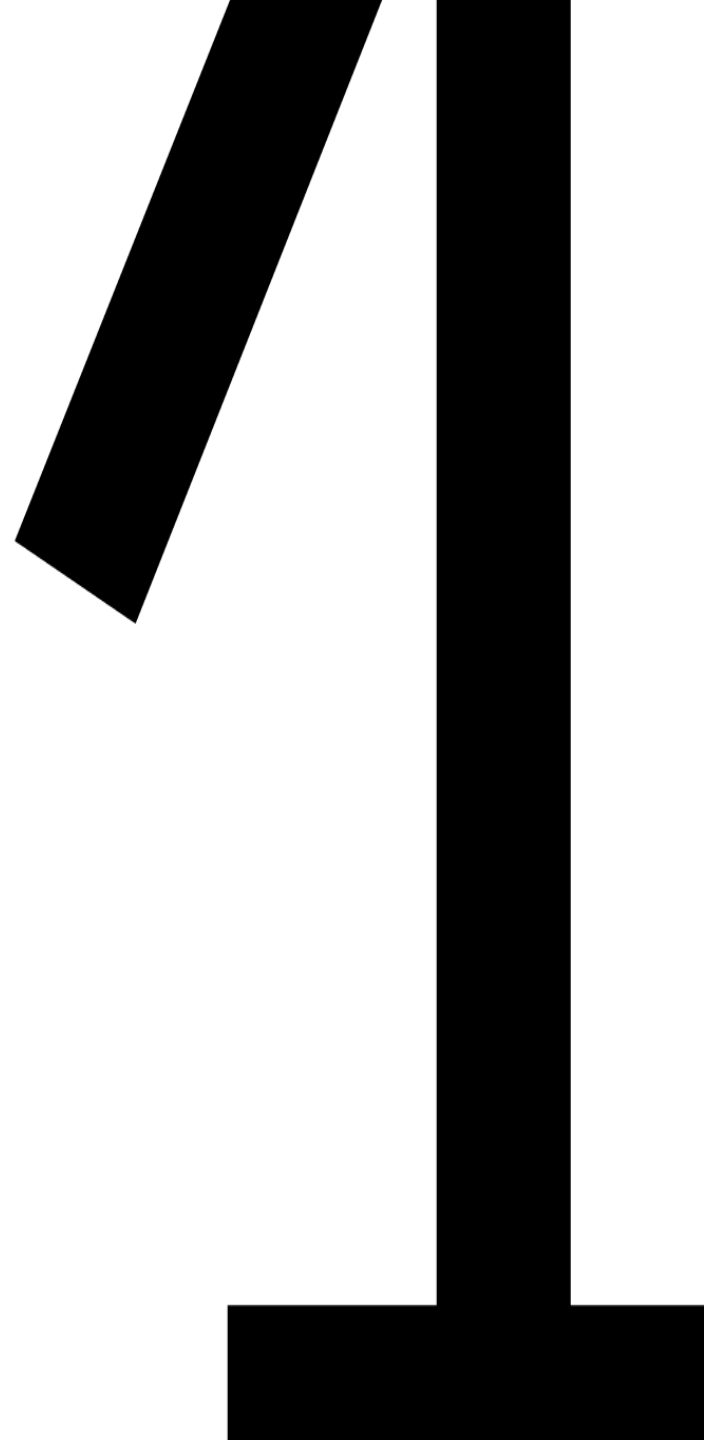
9. XML en JSON

10. Threads

11. Synchronization

12. Concurrency





Reflection

Agenda

1. Deel 1: Reflection

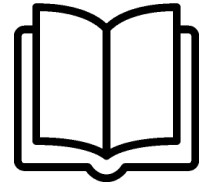
- Wat is reflection?
- De klasse Class
- Voorbeelden van gebruik
- Bedenkingen bij reflection



2. Deel 2: Annotations

- Inleiding
- Standaard annotaties
- Custom annotaties
- Meta-annotaties

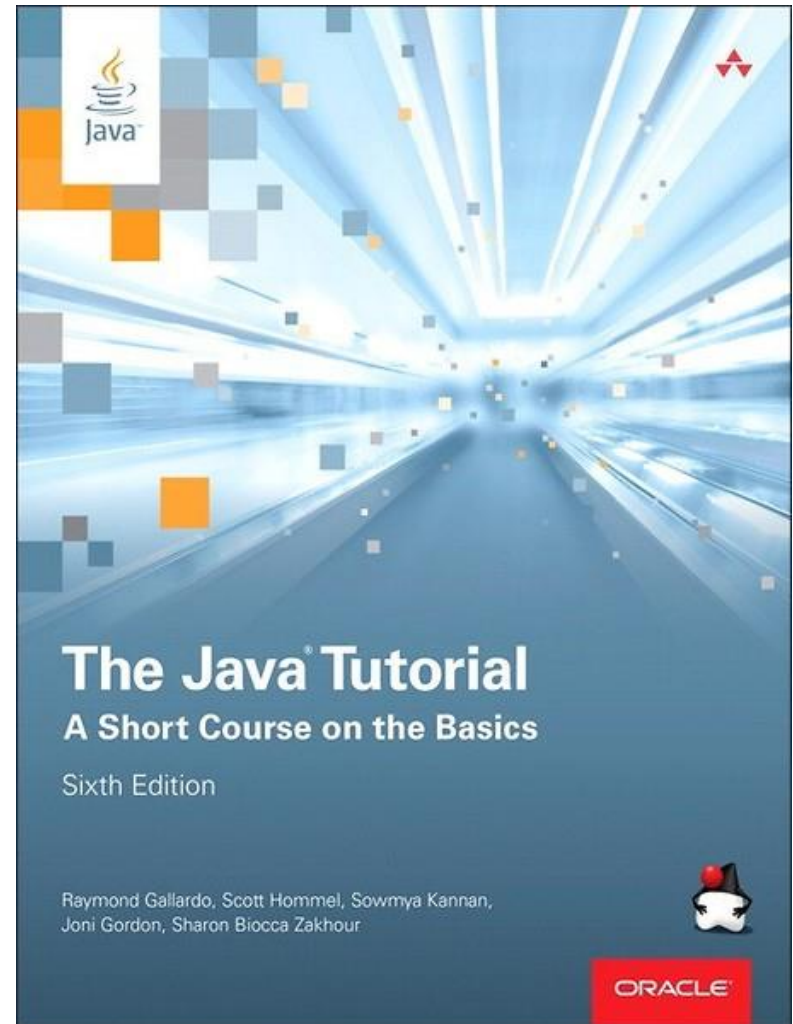
Ondersteunend materiaal



Java Tutorial van Oracle:

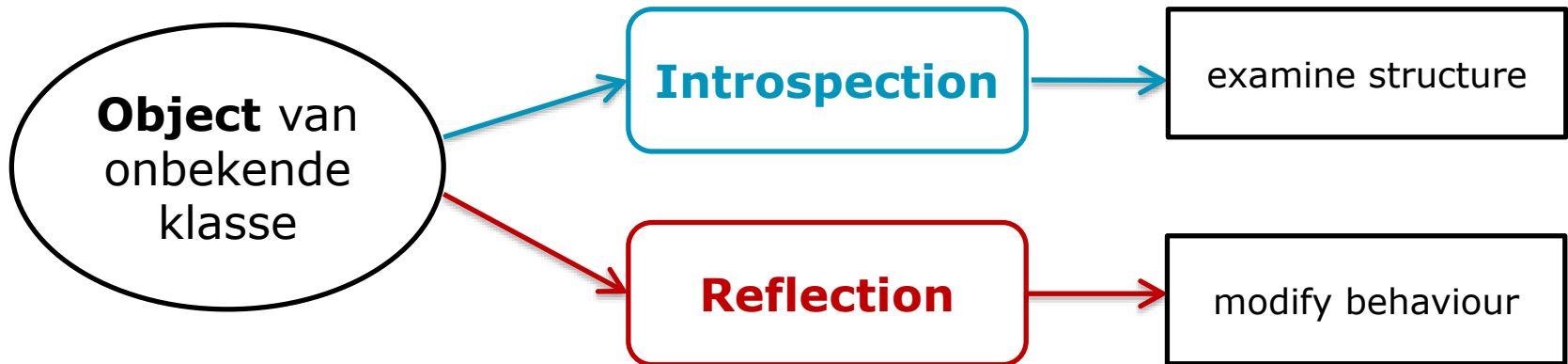
[The Reflection API.](#)

→ Lessons: Classes en Members



Reflection

- **Reflection** is a process of **examining** or **modifying** the run time behavior of a class at run time.
- Dus 2 aspecten zijn mogelijk:



Wat is reflection?

- Reflection wordt oa gebruikt door:
 - IDE (IntelliJ, Eclipse, NetBeans enz...)
 - Class browsers
 - Debuggers
 - Test Tools (zoals JUnit → zie volgende week)
- Met reflection kan je:
 - een klasse instantiëren zonder haar naam te kennen
 - alle methoden van een klasse opvragen
 - alle attributen benaderen (jawel; zelfs de private)
 - annotations opvragen (→ zie volgende deel)

Welke klasse?

- Met de **instanceof** operator kan je testen of een object tot een bepaalde klasse behoort



```
Rectangle rect = new Rectangle();  
System.out.println(rect instanceof Object); //true  
System.out.println(rect instanceof Rectangle); //true
```

→ Je moet wel de klasse die je wil testen op voorhand kennen...

- Kan je de klasse van een object ook gewoon opvragen?

→ Ja! De klasse **Object** heeft een methode **getClass()**

```
Class clazz = rect.getClass();  
System.out.println(clazz.getName());  
//be.kdg.shapes.Rectangle
```


Class objecten

- Andere manieren om een Class object te verkrijgen:

- Op basis van de klasse:

```
Class clazz = Rectangle.class;  
Class primitive = int.class;
```

.class werkt ook voor
primitieve types!

- Op basis van een String met de (volledige) naam:

```
Class clazz = Class.forName("be.kdg.shapes.Rectangle");
```

Dit werkt NIET voor
primitieve types!

Reflection in Java

- De klasse `java.lang.Class` voorziet vele methoden om het runtime gedrag van een klasse te onderzoeken/beïnvloeden en metadata te verkrijgen.
- De package `java.lang.reflect` voorziet een aantal handige klassen om reflection toe te passen.

De klasse Class

- Een greep uit de belangrijkste methoden van de klasse `java.lang.Class`:
 - `getName`
 - `newInstance`
 - `isInterface`
 - `getSuperClass`
 - `getDeclaredFields`
 - `getDeclaredMethods`
 - `getDeclaredConstructors`
- We bespreken hiervan enkele voorbeelden op de volgende slides...



De modelklasse Student

```
public class Student {  
    private final int studNr;  
    private String naam;  
    private LocalDate geboorteDatum;
```

final: komt binnen via constructor; nadien niet meer wijzigbaar

```
    public Student(int studNr, String naam, LocalDate geboorteDatum) {  
        this.studNr = studNr;  
        this.naam = naam;  
        this.geboorteDatum = geboorteDatum;  
    }
```

```
    public Student() {  
        this(0, "Dummy", LocalDate.now());  
    }
```

//getters en setters...

@Override

```
    public String toString() {  
        DateTimeFormatter shortFormatter =  
            DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);  
        return String.format("%s (%s) studNr: %d", naam,  
            shortFormatter.format(geboorteDatum), studNr);  
    }
```

getName

```
Student student = new Student(123456, "Igor De Verschrikkelijke",  
                               LocalDate.of(1995, 9, 18));  
  
System.out.printf("Volledige naam van de klasse: %s\n",  
                  student.getClass().getName());  
  
Integer intObj = new Integer(123);  
  
System.out.printf("Volledige naam van de klasse: %s\n",  
                  intObj.getClass().getName());
```

getName geeft de *fully qualified name* van de klasse
getSimpleName geeft de korte klassenaam

```
/* OUTPUT: */  
Volledige naam van de klasse: model.Student  
Volledige naam van de klasse: java.lang.Integer
```



getMethods / getDeclaredMethods

```
public static void printMethods(Object obj) {  
  
    Class aClass = obj.getClass();  
    System.out.printf("%d methoden gevonden in de klasse %s:\n",  
        aClass.getDeclaredMethods().length,  
        aClass.getSimpleName());  
  
    for (Method method : aClass.getDeclaredMethods()) {  
        System.out.printf("\t%s (returns: %s, parameters: %d)\n",  
            method.getName(),  
            method.getReturnType(),  
            method.getParameterCount());  
    }  
}
```

getMethods geeft alle
(ook dus de overgeërfde
methoden)
getDeclaredMethods
geeft enkel de niet-
overgeërfde methoden

```
/* OUTPUT: */
```

```
6 methoden gevonden in de klasse Student:
```

```
toString (returns: class java.lang.String, parameters: 0)  
getStudNr (returns: int, parameters: 0)  
getGeboorteDatum (returns: class java.time.LocalDate, parameters: 0)  
setGeboorteDatum (returns: void, parameters: 1)  
getNaam (returns: class java.lang.String, parameters: 0)  
setNaam (returns: void, parameters: 1)
```



getDeclaredMethods

<u>Class</u> API	Inherited members?	Private members?
<u>getDeclaredMethods()</u>	no	yes
<u>getMethods()</u>	yes	no

- Enkelvoudsvorm: `getMethod` / `getDeclaredMethod`
 - Geeft één methode met gegeven naam en parameter types

```
Method method =  
    aClass.getMethod("setGeboorteDatum", LocalDate.class);
```

- `getDeclaredMethod` geeft géén inherited methoden en wél private methoden

getConstructors / getDeclaredConstructors

```
public static void printConstructors(Object obj) {
    Class aClass = obj.getClass();
    System.out.printf("%d constructors gevonden in de klasse %s:\n",
        aClass.getDeclaredConstructors().length,
        aClass.getName());
    for (Constructor constructor : aClass.getDeclaredConstructors()) {
        System.out.printf("\t%s (parameters: %d)\n",
            constructor.getName(), constructor.getParameterCount());
    }
}
```

getConstructors geeft enkel de public constructors
getDeclaredConstructors geeft ALLE constructors

```
/* OUTPUT: */
2 constructors gevonden in de klasse model.Student:
    model.Student (parameters: 3)
    model.Student (parameters: 0)
```



newInstance / invoke



```
public static void makeAlive(Class aClass) {
    try {
        System.out.printf("Nieuw object instantiëren van de klasse %s:\n",
            aClass.getName());
        Object object = aClass.newInstance(); //default constructor
        System.out.println(object.toString());
        for (Method method : aClass.getDeclaredMethods()) {
            if (method.getName().startsWith("get")) { //enkel getters
                Object result = method.invoke(object);
                System.out.printf("\tResult van %s: %s\n",
                    method.getName(), result);
            }
        }
    } catch (...) {
        e.printStackTrace();
    }
}
```

newInstance roept een specifieke constructor op (let op de parameter!)
invoke roept een specifieke methode op (let op de parameter!)

```
/* OUTPUT: */
Nieuw object instantiëren van de klasse model.Student:
Dummy (°13/09/16) studNr: 0
    Result van getStudNr: 0
    Result van getGeboorteDatum: 2016-09-13
    Result van getNaam: Dummy
```

getFields / getDeclaredFields

```
public static void examineFields(Object object) {  
    try {  
        Class aClass = object.getClass();  
        System.out.printf("Fields van %s:\n", aClass.getName());  
        for (Field field : aClass.getDeclaredFields()) {  
            System.out.printf("\n%s\" van het type: %s\n",  
                               field.getName(),  
                               field.getType());  
            field.setAccessible(true); //toegang tot private fields!  
            System.out.printf("\ntoegang: %s \n\ntwaarde: %s\n",  
                               Modifier.toString(field.getModifiers()),  
                               field.get(object));  
        }  
    }  
}
```

```
/* OUTPUT: */
```

```
Fields van model.Student:
```

```
"studNr" van het type: int
```

```
    toegang: private final
```

```
    waarde: 123456
```

```
"naam" van het type: class java.lang.String
```

```
    toegang: private
```

```
    waarde: Igor De Verschrikkelijke
```

```
"geboorteDatum" van het type: class java.time.LocalDate
```

```
    toegang: private
```

```
    waarde: 1995-09-18
```

getDeclaredFields
geeft ALLE attributen
getFields geeft enkel
de public attributen



Democode: 01_Reflection

Toegang tot fields



```
public static void changeByReflection(Object object) {
    System.out.println("private fields wijzigen!!!");
    System.out.println("VOOR reflection: " + object);
    try {
        Class aClass = object.getClass();
        Field field = aClass.getDeclaredField("studNr");
        field.setAccessible(true);
        field.set(object, 666);
        field = aClass.getDeclaredField("naam");
        field.setAccessible(true);
        field.set(object, "Mephisto");
        //...
        System.out.println("NA reflection: " + object);
    } catch (...) {
        e.printStackTrace();
    }
}
```

setAccessible verleent toegang tot een private attributen
set wijzigt de waarde van het attribuut

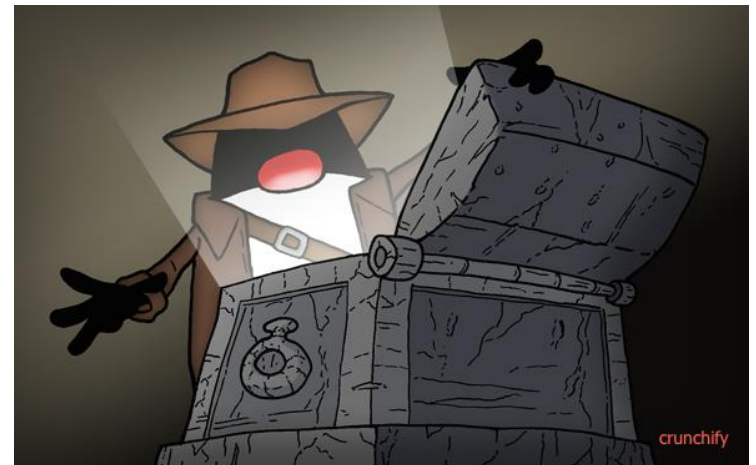
```
/* OUTPUT: */
private fields wijzigen!!!
VOOR reflection: Igor De Verschrikkelijke (°18/09/95) studNr: 123456
NA reflection: Mephisto (°13/09/16) studNr: 666
```



Bedenkingen bij reflection

Reflection is bijzonder krachtig, maar indien mogelijk te vermijden omwille van:

- Performantie
 - Ongeveer 10x trager dan gewone code
- Security
 - In een beperkte context (sandbox) is er mogelijk geen toelating voor reflection
- Inkapseling
 - toegang tot private velden en methoden maakt de klasse kwetsbaar



Opdrachten



- Groeiproject
 - module 3
(deel 1 en 2: "Reflection")



- Opdrachten op BB
 - RentCar



1. Before Breakfast

alliteration

W HERE'S Papa going with that ax?" said Fern to her mother as they were setting the table for breakfast.

"Out to the hoghouse," replied

Mrs. Arable. "Some pigs were born last night."

"I don't see why he needs an ax," continued Fern, who was only eight.

"Well," said her mother, "one of the pigs is a runt. It's very small and weak, and it will never amount to anything. So your father has decided to do away with it."

"Do away with it?" shrieked Fern. "You mean kill it? Just because it's smaller than the others?"

Mrs. Arable put a pitcher of cream on the table.

"Don't yell, Fern!" she said. "Your father is right. The pig would probably die anyway."

Fern pushed a chair out of the way and ran outdoors. The grass was wet and the earth smelled of springtime. Fern's sneakers were sopping by the time she caught up with her father.

Sense

plowable

nature

death

plant

killing

goes with

eating in

a farm

spring pig

@nnotations

Agenda

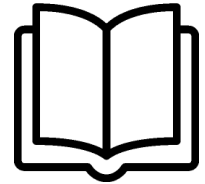
1. Deel 1: Reflection

- Wat is reflection?
- De klasse Class
- Voorbeelden van gebruik
- Bedenkingen bij reflection



2. Deel 2: Annotations

- Inleiding
- Standaard annotaties
- Custom annotaties
- Meta-annotaties



E-book: "Annotation Types" p.229 ev
(The Java Language Specification, Eighth Edition)

- Java Code Geeks Annotation tutorial
(<https://www.javacodegeeks.com/2014/11/java-annotations-tutorial.html>)



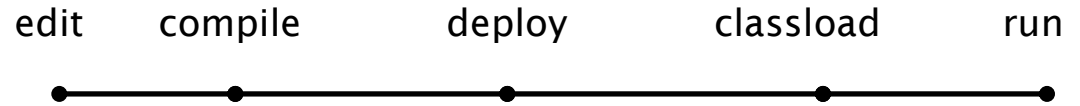
Annotations

- Om **metadata** (= data over data) in de broncode toe te voegen
- Worden door compiler herkend en mee opgenomen in bytecode (<> commentaar)
- Te herkennen door @ als eerste teken
- Vroeger alleen tags voor Javadoc, vanaf JDK5 ook in bron code
- Syntax en type controle

Waarom annotaties?

- **Metadata** toevoegen in de code
 - installatie- of configuratie-instructies
 - info voor de compiler
 - Alternatief voor XML configuratie bestanden
- Veel toegepast in **frameworks**
(JUnit, Hibernate, Spring, ...)

Gebruik van annotaties



–**Edit:**

- informatie voor ontwikkelaar
- Assistentie door IDE

–**Compile:**

- Extra checks door compiler
- Instructies over te genereren code

–**Deploy:** instructies voor installatie tools

–**Classload:** als de code geladen wordt, kunnen extra instructies toegevoegd worden

–**Run:** via reflection kan je annotations in code detecteren en actie ondernemen

Standaard annotations

- Uit java.lang:

@Override

@Deprecated

@SuppressWarnings

@FunctionalInterface (zie later bij lambda's)

@SafeVarargs

@Override

- De **@Override** annotation → aanduiding dat een methode van de superklasse overschreven wordt (= extra controle op compiler-niveau)
- Typisch voorbeeld: override van de methoden **equals**, **hashCode** en **toString** van de klasse **Object**

```
@Override
public boolean equals(Object object) {
    // ...
}
```

@Override

Annotations worden net als klassen en interfaces in een Java bronbestand gedefinieerd.

Je vindt ze in de javadoc van hun package onderaan, in een aparte sectie

java.awt.print
java.beans
java.beans.beancontext
java.io
java.lang
java.lang.annotation
java.lang.annotation.Annotation
java.lang.invoke
java.lang.management
java.lang.ref

Error
ExceptionInInitializerError
IllegalAccessException
IncompatibleClassChangeError
InstantiationError
InternalError
LinkageError
NoClassDefFoundError
NoSuchFieldError
NoSuchMethodError
OutOfMemoryError
StackOverflowError
ThreadDeath
UnknownError
UnsatisfiedLinkError
UnsupportedClassVersionError
VerifyError
VirtualMachineError

Annotation Types

Deprecated
Override
SafeVarargs
SuppressWarnings

Overview Package **Class** Use Tree Depre
Prev Class Next Class Frames No Fram
Summary: Required | Optional Detail: Element

java.lang

Annotation Type Override

```
@Target(value=METHOD)  
@Retention(value=SOURCE)  
public @interface Override
```

Indicates that a method declaration is intended to override a method in a superclass. Compilers are required to generate code for the method.

- The method does override or implement a method in a superclass.
- The method has a signature that is overridden by the method.

Since:

1.5

See *The Java™ Language Specification*:

9.6.1.4 Override

Overview Package **Class** Use Tree Depre
Prev Class Next Class Frames No Fram
Summary: Required | Optional Detail: Element

@Deprecated

De **@Deprecated** annotation → aanduiding dat het gebruik van een methode afgeraden is en door een nieuwe vervangen werd.

(Geeft alleen een compiler warning)

```
/**  
 * Wijzigt de waarde van de x coördinaat.  
 * @param x de coördinaat  
 *  
 * @deprecated gebruik setCoordinaten  
 */
```

@Deprecated

```
public void setX(int x) {  
    this.x = x;  
}
```

```
punt.setX(2);  
punt.setY(3);  
punt.setCoordinaten(2, 3);
```



@SuppressWarnings

De **@SuppressWarnings** annotation → onderdrukt bepaalde warnings

```
@SuppressWarnings ( { "unchecked", "unused" } )  
public void eenMethode () {  
    // ...  
}
```

→ onderdrukt de warnings in verband met het niet gebruiken van generics (**unchecked**) en ongebruikte variabelen (**unused**).

Custom Annotations

- Je kan **ZELF** een annotation maken
- Indeling op basis van parameters:
 - Geen: **Marker** annotation
zoals `@Override`
 - Één: **Single-value** annotation
zoals `@SuppressWarnings("unchecked")`
 - Meerdere: **Full** annotation
zoals `@Test(expected = IOException.class,
timeout = 100)`

Voorbeeld: Marker annotation

```
package p;  
  
@interface MyAnno {}
```

Definitie
(MyAnno.java)



```
import p.MyAnno;  
  
@MyAnno  
class MyClass {}
```

Gebruik
(MyClass.java)



- Definitie lijkt op een interface met '@' ervoor

Voorbeeld: Single-value annotation

```
@interface MaxLength {  
    int value() default 80;  
}
```

Definitie
(MyAnno.java)

```
interface Foo {  
    @MaxLength(25)  
    String getFirstName();  
  
    @MaxLength  
    String getLastName();  
}
```

Of: (value=25)

Gebruik
(MyClass.java)

Defaultwaarde
wordt gebruikt

- Als er slechts één attribuut is, én dat heeft de naam **value**, dan mag je "**value=**" weglaten
- Als alle attributen een **default** hebben, kan je de annotatie gebruiken alsof het een marker is

Voorbeeld: Full annotation

```
@interface Since {  
    int major();  
    int minor() default 0;  
    String[] authors();  
    String[] reviewers();  
}
```

Definitie
(MyAnno.java)

Array attributen

```
@Since(major = 3,  
        authors = {"Jan", "Piet", "Joris"},  
        reviewers = "Corneel"  
)  
class MyClass {}
```

Gebruik
(MyClass.java)

Array literals

- Hier worden **Array attributen** gebruikt:
 - Array literal: Meerdere waarden tussen {...}
 - Je mag ook een enkele waarde aan een array toekennen

Geneste annotaties

- Je mag elk type gebruiken voor een annotatie attribuut, ook een annotatie type:

```
@interface Column {  
    String name();  
    MaxLength check();  
}
```

Definitie
(MyAnno.java)

```
class Artikel{  
    @Column(name="COMM", check=@MaxLength(120))  
    String commentaar;  
}
```

Gebruik
(MyClass.java)

Geneste annotaties

- Je kan ook een array van geneste annotaties hebben:

```
@interface Table {  
    Column[] value();  
}
```

Definitie
(MyAnno.java)

```
@Table{  
    @Column(name="MERK", check=@MaxLength(40))  
    @Column(name="BESCHRIJVING", check=@MaxLength(120))  
}  
class Product{... }
```

Gebruik
(MyClass.java)

Annotations annoteren

```
@Retention (RetentionPolicy.RUNTIME)  
@Target ({ ElementType.FIELD, ElementType.PARAMETER })  
@Documented  
@Inherited  
@interface MyAnno { }
```

Gedefinieerd in java.lang.annotation

@Retention

- SOURCE: in source, maar de compiler zet de annotatie niet in de bytecode
- CLASS (default): in bytecode, maar als de code geladen wordt wordt de annotatie niet mee omgezet in runtime code
- RUNTIME: in source, bytecode en runtime code

Annotaties annoteren

```
@Retention (RetentionPolicy.RUNTIME)  
@Target ({ ElementType.FIELD, ElementType.PARAMETER })  
@Documented  
@Inherited  
@interface MyAnno { }
```

@Target

- Op welke elementen kan MyAnno gebruikt worden?
(Default: all)

@Documented

- Zal MyAnno vermeld worden in de javadoc van de klassen waarin ze gebruikt wordt?
(Default: no)

getDeclaredAnnotations

- Omdat sommige annotations at runtime beschikbaar zijn, kunnen we acties ondernemen via ***reflection***.

```
for (Method method : myClass.class.getDeclaredMethods()) {  
    MyAnno myAnno = method.getDeclaredAnnotation(MyAnno.class);  
    // ...  
}
```

Voorbeeld: Definitie Marker annotation

- We maken zelf een marker annotation met de naam `Unfinished`:

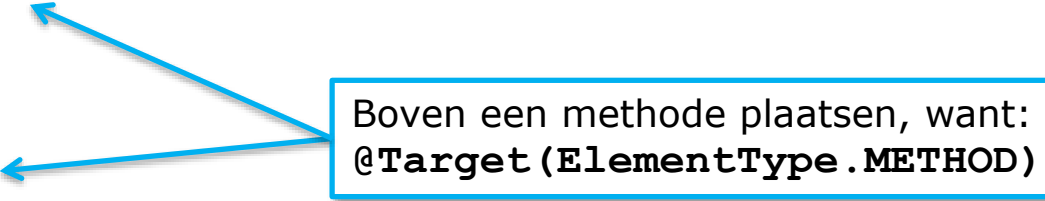
```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Unfinished {
}
```

`@Target` bepaalt dat de annotatie gebruikt zal worden bij een **methode**

`@Retention` bepaalt dat de annotatie **at runtime** beschikbaar moet zijn (want we willen **reflection** toepassen!)

Voorbeeld: **Gebruik** Marker annotation

```
public class Spel {  
    public void startSpel() { /* ... */ }  
  
    @Unfinished  
    public void login(String user, char[] passWord) {  
        // nog uitwerken  
    }  
  
    @Unfinished  
    public List<String> getTopscores() {  
        // nog uitwerken  
        return Collections.emptyList();  
    }  
}
```



Boven een methode plaatsen, want:
@Target(ElementType.METHOD)

Voorbeeld: Reflection op Marker annotation

```
public class DemoMarker {  
    public static void main(String[] args) {  
        for (Method method : Spel.class.getDeclaredMethods()) {  
            Unfinished unfinished = method.getAnnotation(  
                Unfinished.class);  
  
            System.out.print(method.getName());  
            if (unfinished != null) {  
                System.out.println(" --> NOG AFWERKEN!");  
            } else {  
                System.out.println(" --> OK");  
            }  
        }  
    }  
}
```

```
getTopscores --> NOG AFWERKEN!  
login --> NOG AFWERKEN!  
startSpel --> OK
```



Reflection opdracht



- Open de voorbeelden die bij de slides horen. Bekijk de module `02_Deprecated` en de werking ervan.
- Voeg een nieuwe klasse `ReflectionDemo` toe met een `main`, waarin je reflection toepast.
 - Maak een `ArrayList` met de namen van alle methoden van de klassen `Punt` en `RuimtePunt` die *deprecated* zijn.
 - Druk vervolgens de namen van de methoden in de `ArrayList` op één regel af.

```
/* OUTPUT: */  
setY setX setZ
```

Tip: Bekijk goed welke methoden de klasse `Method` bevat.

Voorbeeld: Definitie Single-value annotation

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Unfinished {
    String value();
}
```

Voorbeeld: **Gebruik** Single-value annotation

```
public class Spel {  
    // ...  
  
    public void startSpel() {  
        // ...  
    }  
  
    @Unfinished(value = "Uitwerken login")  
    public void login(String user, char[] passWord) {  
        // nog uitwerken  
    }  
  
    @Unfinished("Uitwerken topscores")  
    public List<String> getTopscores() {  
        // nog uitwerken  
        return Collections.emptyList();  
    }  
}
```

Bij single-value mag je de toekenning weglaten! De naam in de @interface moet dan wel **value** zijn.

Voorbeeld: Reflection op Single-value annotation

```
public class DemoSingleValue {  
    public static void main(String[] args) {  
        for (Method method : Spel.class.getDeclaredMethods()) {  
            Unfinished unfinished = method.getAnnotation(  
                Unfinished.class);  
  
            System.out.print(method.getName());  
            if (unfinished != null) {  
                System.out.println(" --> NOG AFWERKEN: "  
                    + unfinished.value());  
            } else {  
                System.out.println(" --> OK");  
            }  
        }  
    }  
}
```

```
getTopscores --> NOG AFWERKEN: Uitwerken topscores  
startSpel --> OK  
login --> NOG AFWERKEN: Uitwerken login
```



Voorbeeld: Definitie Full-type annotation

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Unfinished {
    public enum Belangrijkheid {
        KRITIEK, BELANGRIJK, GEWOON
    }

    Belangrijkheid belangrijkheid()
        default Belangrijkheid.BELANGRIJK;

    String todo();

    String eindDatum();
}
```

Voorbeeld: Gebruik Full-type annotation

```
public class Spel {
    public void startSpel() {
        // ...
    }

    @Unfinished(belangrijkheid = Unfinished.Belangrijkheid.KRITIEK,
        todo = "Uitwerken login", eindDatum = "01/11/16")
    public void login(String user, char[] passWord) {
        // nog uitwerken
    }

    @Unfinished(belangrijkheid = Unfinished.Belangrijkheid.GEWOON,
        todo = "Uitwerken topscores", eindDatum = "01/12/16")
    public List<String> getTopscores() {
        // nog uitwerken
        return Collections.emptyList();
    }
}
```

Voorbeeld: Reflection op Full-type annotation

```
public class DemoFullType {  
    public static void main(String[] args) {  
        for (Method method : Spel.class.getDeclaredMethods()) {  
            Unfinished unfinished = method.getAnnotation( Unfinished.class);  
            System.out.print(method.getName());  
            if (unfinished != null) {  
                System.out.println(" --> NOG AFWERKEN: " +unfinished.todo());  
                System.out.println("\tPrioriteit: "  
                    + unfinished.belangrijkheid());  
                System.out.println("\tEinddatum: "+ unfinished.eindDatum());  
            } else {  
                System.out.println(" --> OK");  
            }  
        }  
    }  
}
```

```
startSpel --> OK  
login --> NOG AFWERKEN: Uitwerken login  
    Prioriteit: KRITIEK  
    Einddatum: 01/11/16  
getTopscores --> NOG AFWERKEN: Uitwerken topscores  
    Prioriteit: GEWOON  
    Einddatum: 01/12/16
```

Om de waarde van een
annotatie op te vragen,
roep je ze aan zoals een
methode



Democode: 05_FullType

Hoe werden de standaard annotations zelf gedefinieerd?

`java.lang.Override`

```
@Target(value=METHOD)
@Retention(value=SOURCE)
public @interface Override
```

`java.lang.Deprecated`

```
@Documented
@Retention(value=RUNTIME)
public @interface Deprecated
```


`java.lang.SuppressWarning`

```
@Target(value={TYPE, FIELD, METHOD, PARAMETER,
CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings
```

Reflectievoorbeeld

- De annotation DoeDeGroeten:

```
@Target({ElementType.FIELD,  
        ElementType.METHOD,  
        ElementType.TYPE,  
        ElementType.CONSTRUCTOR})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface DoeDeGroeten {  
    public String value();  
}
```



Meerdere targets
mogelijk; array via
accolades

Reflectievoorbeeld (2)

- De klasse `Hallo`:

```
@DoeDeGroeten("Hello, class!")
public class Hallo {

    @DoeDeGroeten("Hello, field!")
    public String groetToestand;

    @DoeDeGroeten("Hello, constructor!")
    public Hallo() {
    }

    @DoeDeGroeten("Hello, method!")
    public void groet() {
    }
}
```

Reflectievoorbeeld (3)

```
public class ReflectieDemo {  
    public static void main(String[] args) throws Exception {  
        Class<Hallo> clazz = Hallo.class;  
        System.out.println(clazz.getAnnotation(DoeDeGroeten.class));  
        Constructor<Hallo> cstr = clazz.getConstructor(  
                                                    (Class[]) null);  
        System.out.println(cstr.getAnnotation(DoeDeGroeten.class));  
        Method method = clazz.getMethod("groet");  
        System.out.println(method.getAnnotation(DoeDeGroeten.class));  
        Field field = clazz.getField("groetToestand");  
        System.out.println(field.getAnnotation(DoeDeGroeten.class));  
    }  
}
```

```
@annotations.DoeDeGroeten(value=Hello, class!)  
@annotations.DoeDeGroeten(value=Hello, constructor!)  
@annotations.DoeDeGroeten(value=Hello, method!)  
@annotations.DoeDeGroeten(value=Hello, field!)
```



Opdrachten



- Groeiproject
 - module 3
(deel 3: "Annotations")
- Opdrachten op BB
 - Reflection
 - Kleurannotation
 - Annotation
- Zelftest!

