



Odisee
DE CO-HOGESCHOOL

Software development lifecycle



Maarten Troost – Jens Baetens



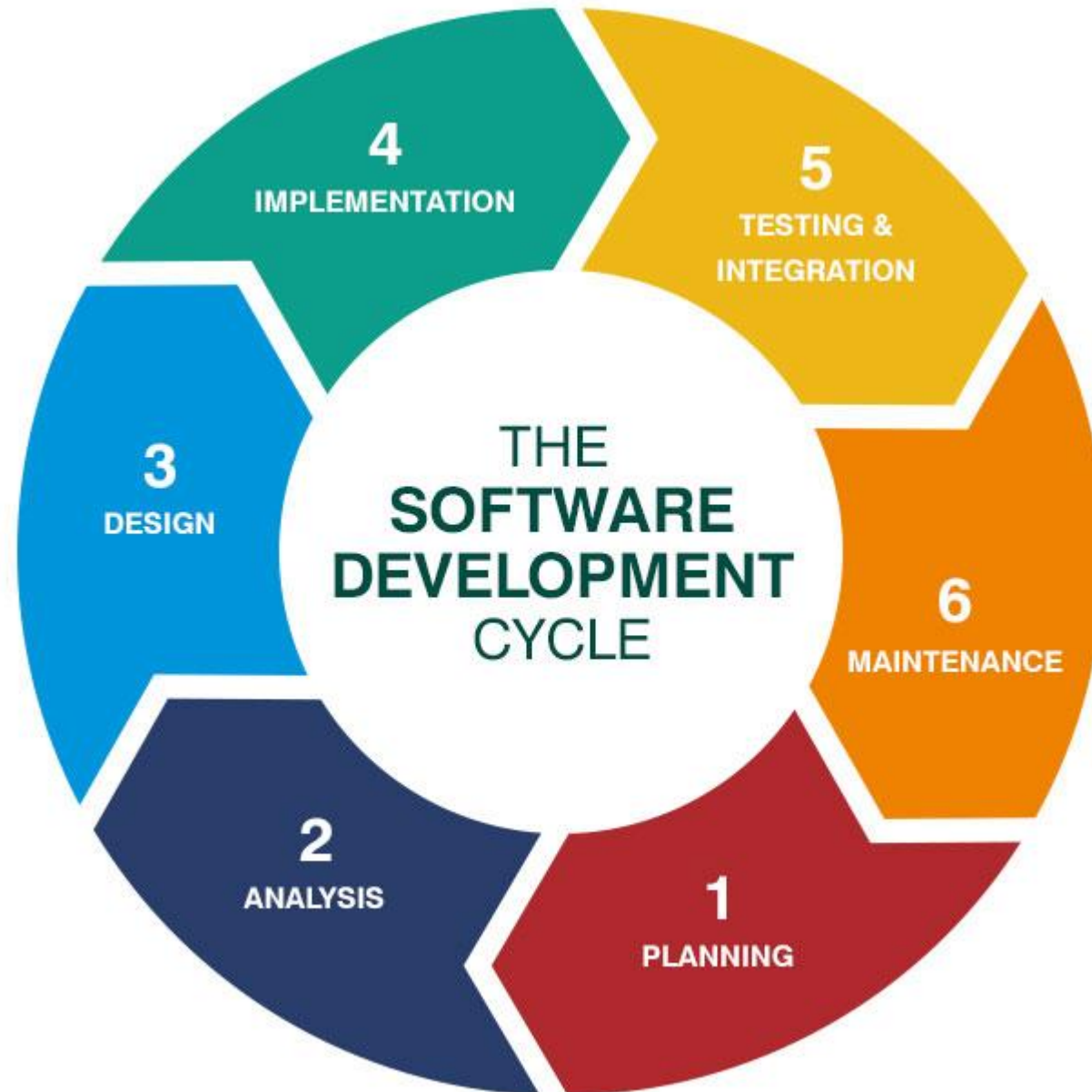
Project Lifecycle



Wat zit in de lifecycle?

Welke fasen doorloopt het ontwikkelen van een software/hardware product?

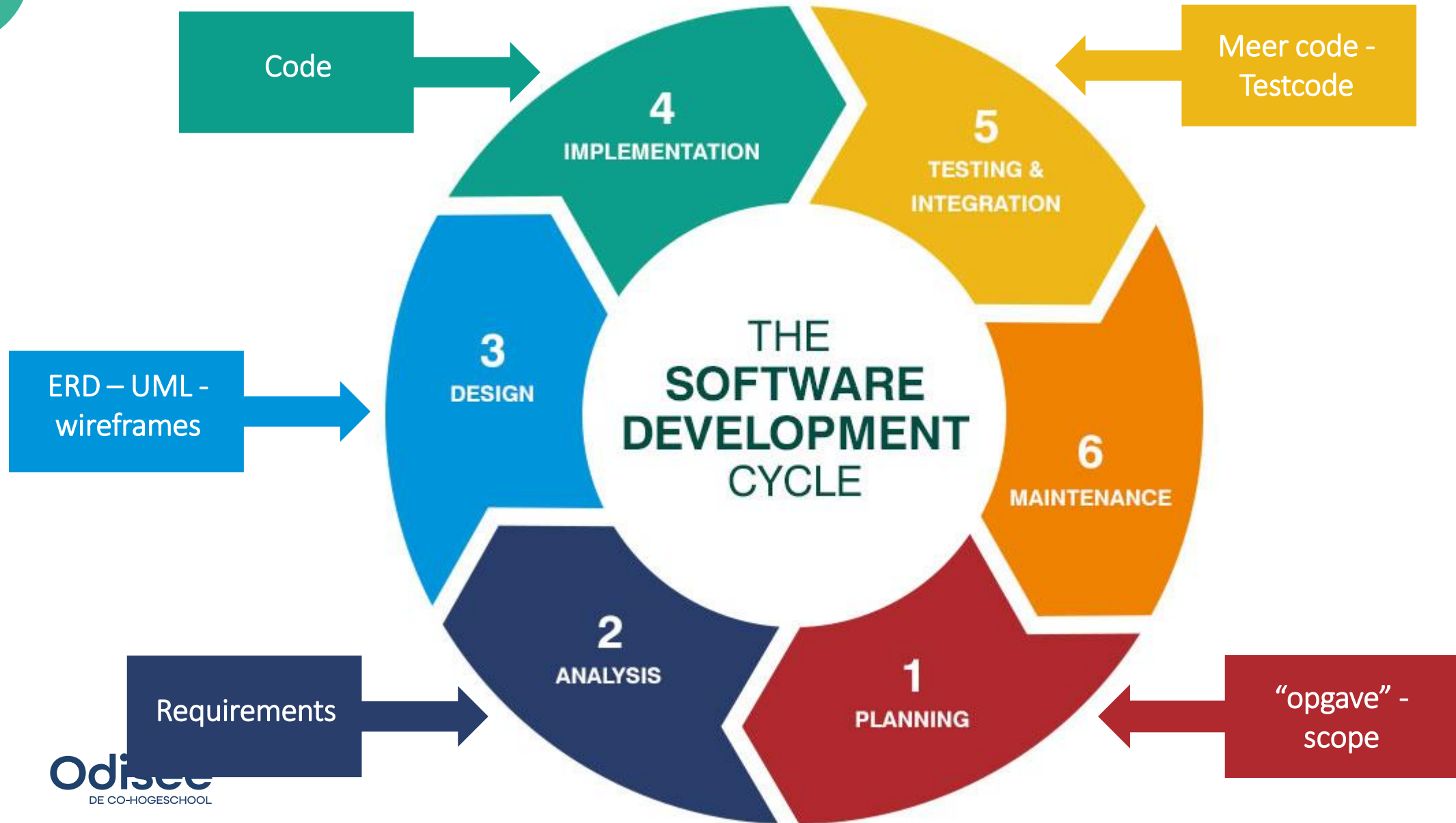
Wat zit in de lifecycle?



Wat wil dit zeggen voor de project lifecycle?

- ▣ Doorgronden van de probleemstelling => analyse
- ▣ Opstellen van lijst requirements => analyse
- ▣ Toekennen van de verantwoordelijkheden => design
- ▣ Herhaal:
 - ▬ JUnit test maken (op basis van de lijst)
 - ▬ Schrijf code (eerst het skelet, daarna de details)
 - ▬ Refactor
 - ▬ Javadoc-documentatie schrijven
- ▣ En daarna ... de GUI dus

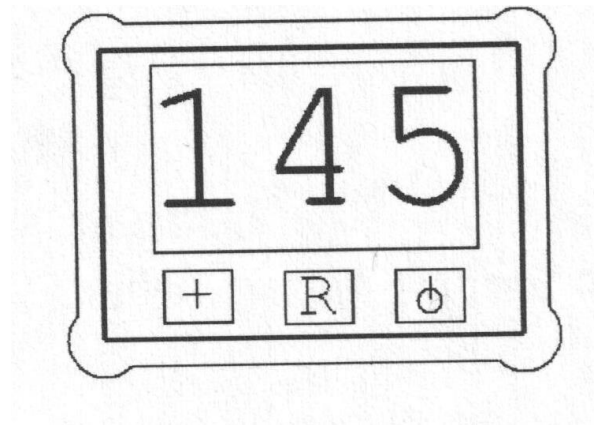
Wat zit in de lifecycle?





Requirements

Hoe beschrijf je een applicatie?



Methode 1: Beschrijving in natuurlijke taal

- ▣ Een teller dient om het aantal malen van een bepaalde gebeurtenis bij te houden.
- ▣ Hij kan in- en uitgeschakeld worden.
- ▣ Indien een teller ingeschakeld is, dan bezit hij de mogelijkheid de huidige stand mee te delen.
- ▣ Nog steeds als hij ingeschakeld is, kan de teller met 1 verhoogd worden.
- ▣ Hij kan ook terug op 0 gezet worden.
- ▣ Indien een teller uitgeschakeld is, geeft de display steeds een 0 weer en reageren de knoppen om te resetten en de waarde te verhogen niet.



Methode 2: Grafische beschrijving

▣ Bijvoorbeeld via UML

- Unified Modelling Language

Teller	
-	waarde: int
-	tellerStaatAan: boolean
+	Teller()
+	increment(): void
+	reset(): void
+	getWaarde(): int
+	zetTellerAan(): void
+	zetTellerUit(): void
+	staatTellerAan(): boolean

Methode 3: Beschrijving in programmeertaal

```
package be.odisee;

/**
 * Een eenvoudige teller
 * @author Jens Baetens
 */
public class Teller {

    private int waarde;
    private boolean tellerStaatAan;

    /**
     * Default constructor: een Teller-object wordt aangemaakt
     * met waarde 0 en is uitgeschakeld.
     */
    public Teller(){
        this.waarde = 0;
        this.tellerStaatAan = false;
    }

    /**
     * Vermeerder waarde van Teller met 1.
     * Heeft geen effect als de teller uit staat.
     */
    public void increment(){
        if (tellerStaatAan){
            this.waarde++;
        }
    }
}
```

```
/**
 * Zet waarde van Teller op 0.
 * Heeft geen effect als de teller uit staat.
 */
public void reset(){
    if (tellerStaatAan){
        this.waarde = 0;
    }
}

/**
 * Return waarde van de Teller
 * Indien de teller uitstaat, geef nul terug.
 */
public int getWaarde(){
    if (tellerStaatAan) {
        return this.waarde;
    } else {
        return 0;
    }
}

/** Zet teller aan en zet waarde op 0 */
public void zetTellerAan(){
    this.tellerStaatAan = true;
    this.waarde = 0;
}

/** Zet teller uit en zet waarde op 0 */
public void zetTellerUit(){
    this.tellerStaatAan = false;
    this.waarde = 0;
}

/** Controleer of de teller aanstaat */
public boolean staatTellerAan(){
    return this.tellerStaatAan;
}
```



Requirements

- ▣ Natuurlijke taal
 - ▣ Beschrijft de functionaliteiten
 - ▣ Omschrijft de verantwoordelijkheden
-
- ▣ Pro: verstaanbaar voor iedereen
 - ▣ Con: niet precies, interpretatie mogelijk, domeinkennis nodig
-
- ▣ Het formaliseren van requirements, zodat ze minder interpreteerbaar worden, komt aan bod in Business Analysis.

Requirements

High level requirements

- ▣ Bevatten weinig detail
- ▣ Voorbeelden:
 - Een teller kan verhoogd worden of gereset worden.
 - Een teller kan aan- of uitstaan waardoor het niet kan veranderen.

(Low level) Requirements

- ▣ Vermelden alle functionaliteiten in detail
- ▣ Voorbeelden:
 - De ingeschakelde teller met 1 verhoogd worden.
 - Hij kan ook terug op 0 gezet worden.

Welke kenmerken heeft een goed requirement?

Wat zijn goede requirements? Wat kan beter?

▣ Voorbeeld 1:

- ▬ Een ticket klasse kan valideren of een ticket geldig is (true of false).

▣ Voorbeeld 2:

- ▬ Bepaal met de zeef van Erathostenes of een getal een priemgetal is.

▣ Voorbeeld 3:

- ▬ Het printen moet snel gebeuren, voor de gebruiker het toestel verlaat.

Requirements

- ▣ Lijst
- ▣ Korte zinnen
- ▣ Opsplitsen per onderwerp
- ▣ Moet verifieerbaar/meetbaar zijn
- ▣ Vanuit het gezichtspunt van de gebruiker. Dit kan een persoon maar ook andere software (vb API gebruiker) zijn.
- ▣ Uitputtend: alles vermelden
- ▣ Beschrijft GEEN implementatie: niet vermelden van hoe dit gerealiseerd wordt maar enkel uiterlijke kenmerken

Wat zijn goede requirements? Wat kan beter?

▣ Voorbeeld 1:

- ▬ Een ticket klasse kan valideren of een ticket geldig is (true of false).
Een ticket kan op geldigheid worden gevalideerd.

▣ Voorbeeld 2:

- ▬ Bepaal met de zeef van Erathostenes of een getal een priemgetal is.
- ▬ Bepaal of een getal een priemgetal is.

▣ Voorbeeld 3:

- ▬ Het printen moet snel gebeuren, voor de gebruiker het toestel verlaat.
- ▬ De print activiteit moet afgerond zijn binnen de 5 seconden na de bevestiging.

Van requirements naar verantwoordelijkheden

- ▣ Koppel aan elke requirement 1 klasse
- ▣ Of meerdere klassen indien er een opsomming in het requirement staat
- ▣ Indien mogelijk, bepaal de naam van de methode welke het requirement zal implementeren
- ▣ Een goede klasse of methodenaam vertelt welk requirement deze invult
- ▣ Denk aan het Single Responsibility Principle! Splits alle verantwoordelijkheden op.

Van requirements naar verantwoordelijkheden

- ▣ Een ticket kan op geldigheid worden gevalideerd.
=> Class ticket, method isGeldig()
OF => Class ticketValidator
- ▣ Bepaal of een getal een priemgetal is.
=> PrimeNumber.isPrime(Integer number)
OF => Integer.isPrime()
- ▣ De print activiteit moet afgerond zijn binnen de 5 seconden na de bevestiging.
=> Bestelling.print()
OF => PrintService.acceptJob(bestelling)

Oefening van requirements naar verantwoordelijkheden

▣ Requirements:

a) We simuleren een broodrooster welke gedurende 1 minuut roostert als we de broodrooster starten.

b) De broodrooster kan 2 sneden tegelijk roosteren

c) Een temperatuurregelknop op het apparaat bepaalt hoeveel weerstand het apparaat levert

▣ Bepaal welke klasse / variabele / methode verantwoordelijk is.

Oefening van requirements naar verantwoordelijkheden

▣ Requirements:

a) We simuleren een broodrooster welke gedurende 1 minuut roostert als we de broodrooster starten.

=> klasse Broodrooster, methode Broodrooster.start()

b) De broodrooster kan 2 sneden tegelijk roosteren

=> Integer Broodrooster.aantalSneden

c) Een temperatuurregelknop op het apparaat bepaalt hoeveel weerstand het apparaat levert

=> Broodrooster.weerstand



Documentatie

Documentatie - Javadoc

- ▣ Platform voor genereren van documentatie voor Java applicatie
 - ▬ Zelfde stijl / manier voor alle applicaties
 - ▬ Gegenerend op basis van documentatie in code
- ▣ Informatie over hoe de applicatie / klassen te gebruiken
- ▣ Voorbeeld: (google javadoc String)
<https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/String.html>



Manieren voor documentatie

- ▣ `// ...` voor single line
- ▣ `/* ... */` voor commentaar tussen code lijnen
 - ▬ Voor extra toelichting van de code maar niet in de documentatie
- ▣ `/** ... */`
 - ▬ Commentaar wordt opgevangen bij genereren van documentatie
 - ▬ Boven een methode / klasse
 - ▬ Voeg minstens een beschrijving van klasse / methode / parameters / return waarde toe



Voorbeeld voor PetRock

```
/**
 * Deze klasse stelt een huisdier-steen voor met een naam en of de steen gelukkig is
 * @author Jens Baetens
 * @version 0.1
 */
public class PetRock {

    private String name;
    private boolean happy = false;

    /**
     * Constructor voor het aanmaken van het object met een naam, standaard geluk is false
     * @param name Naam van de steen
     */
    public PetRock(String name) {
        this.name = name;
    }

    /**
     * Getter voor de naam
     * @return de naam
     */
    public String getName() {
        return name;
    }
}
```

Genereren documentatie

▣ Javadoc command line

```
C:\Users\jens.baetens3\OneDrive - ODISEE\SEF\Leerstof\Leerstof\Week 2\Code\src>javadoc -author -version -d ../documentation/ -subpackages .
```

Pad waar code staat

Commando

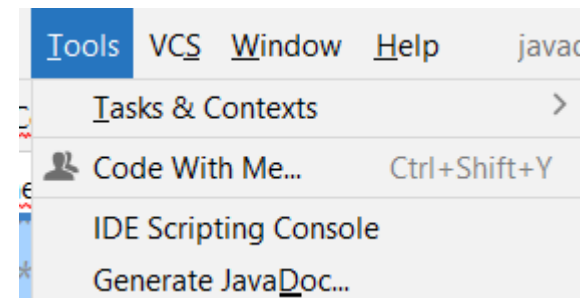
Waar moet html staan

In welke folder moet er gezocht worden

▣ IntelliJ IDEA:

- Maak een directory javadoc aan in het project en genereer de documentatie via Tools > Generate JavaDoc.

▣ Zie ook Toledo > inhoud > Hoe java documentatie schrijven



Genereren documentatie

- ▣ Javadoc command line
- ▣ Via IntelliJ

The screenshot shows the 'Generate Javadoc' dialog in IntelliJ IDEA. It is configured with the following settings:

- Generate Javadoc Scope:**
 - ☒ Whole project
 - ☐ Custom scope: All Places
 - ☒ Include test sources
- Generation Options:**
 - ☐ Include JDK and library sources in -sourcepath
 - ☐ Link to JDK documentation (use -link option)
- Output directory:** eerstof\Leerstof\Week 2\Code\documentatie2
- Visibility:** A vertical slider on the left is set to 'protected'.
- Content Options:**
 - ☒ Generate hierarchy tree
 - ☒ Generate navigation bar
 - ☒ Generate index
 - ☒ Separate index per letter
 - ☐ @use
 - ☒ @author
 - ☒ @version
 - ☒ @deprecated
 - ☒ Depreciated list
- Locale:** (empty text field)
- Other command line arguments:** (empty text field)
- Maximum heap size (Mb):** (empty text field)
- ☒ Open generated documentation in browser

Buttons at the bottom: ? (help), OK, and Cancel.

Genereren documentatie

be.odisee

Class PetRock

java.lang.Object
be.odisee.PetRock

```
public class PetRock  
extends java.lang.Object
```

Deze klasse stelt een huisdier-steen voor met een naam en of de steen gelukkig is

Version:

0.1

Author:

Jens Baetens

Constructor Summary

Constructors

Constructor and Description

PetRock(java.lang.String name)

Constructor voor het aanmaken van het object met een naam, standaard geluk is false

Best practices

- ▣ Zie Toledo > Inhoud > Hoe javadocumentatie schrijven.
- ▣ Alle klassen worden van een Javadoc comment voorzien bovenaan.
- ▣ Ook alle public en protected methoden en variabelen.
- ▣ In de Javadoc comment schrijf je de verantwoordelijkheden neer.
- ▣ Beschrijf alle parameters, return values en exceptions.
- ▣ Vermeld alle beperkingen of kennis welke de gebruiker van je method of class kan nodig hebben.

Javadoc en het codeskelet

- In de Javadoc documenten zie je het codeskelet. Dat is de vorm van de code zonder de uitvoerbare code. Of nog: alle code zonder de body van de methods.

```
/** Deze comment toont de verantwoordelijkheden van de klasse en is deel van het codeskelet
 * Verantwoordelijkheid: een voorbeeld van een codeskelet zijn */
class CodeSkelet {
    //Een private variabele heeft geen javadoc nodig
    private final long myConstant;
    protected final List<CodeBot> botten;

    /** Een constructor wel. Deze initialiseert klassevariabelen
     * @param botten Alle botten waaruit het skelet wordt opgebouwd
     */
    public CodeSkelet(List<CodeBot> botten) {...}

    /** Verantwoordelijkheid is het bepalen welke botten niet structureel nodig zijn
     * @return de overbodige botten */
    public List<CodeBot> overbodigeBotten(...) {...}
}
```

Method
body

Javadoc en het codeskelet

Class CodeSkelet

`java.lang.Object`
`be.odisee.CodeSkelet`

```
public class CodeSkelet  
extends Object
```

Deze comment toont de verantwoordelijkheden van de klasse en is deel van het codeskelet Verantwoordelijkheid: een voorbeeld van een codeskelet zijn

Constructors

Constructor	Description
<code>CodeSkelet(List<be.odisee.CodeBot> botten)</code>	Een public constructor moet wel javaodc krijgen.

Method Details

overbodigeBotten

```
public List<be.odisee.CodeBot> overbodigeBotten()
```

Verantwoordelijkheid is het bepalen welke botten niet structureel nodig zijn

Returns:

de overbodige botten

Oefening documentatie verantwoordelijkheden

a) We simuleren een broodrooster welke gedurende 1 minuut roostert als we de broodrooster starten.

=> klasse Broodrooster, methode Broodrooster.start()

b) De broodrooster kan 2 sneden tegelijk roosteren

=> Integer Broodrooster.aantalSnedes

c) Een temperatuurregelknop op het apparaat bepaalt hoeveel weerstand het apparaat levert

=> Broodrooster.weerstand

Schrijf het codeskelet van deze verantwoordelijkheden inclusief de javadoc

Oefening documentatie verantwoordelijkheden oplossing

▣ Requirements:

a) We simuleren een broodrooster welke gedurende 1 minuut roostert als we de broodrooster starten.

=> klasse Broodrooster, methode Broodrooster.start()

```
/**  
 * Deze klasse stelt een broodrooster voor met 1 knop: een temperatuurregelaar  
 */  
class Broodrooster {  
  
    /**  
     * Req a) Start het roosteren gedurende 1 minuut  
     */  
    public void start() {}  
}
```

Oefening documentatie verantwoordelijkheden oplossing

b) De broodrooster kan 2 sneden tegelijk roosteren

=> Integer Broodrooster.aantalSneden

```
/**  
 * Req b) Bepaalt hoeveel sneden tegelijk geroosterd worden  
 */  
private final Integer aantalSneden=2;
```

c) Een temperatuurregelknop op het apparaat bepaalt hoeveel weerstand het apparaat levert

=> Broodrooster.weerstand

```
/** Req c) De elektrische weerstand zoals bepaald door de temperatuurregelaar */  
private Double weerstand;  
public void setWeerstand(Double weerstand) {}
```

Oefening verantwoordelijkheden tekenen

- a) Wijs de requirements toe aan nieuwe klassen/methods/variabelen. Dit is een onderdeel van een tekenprogramma.
 - 1. Op een doek kunnen verschillende figuren getekend worden: vierkanten en cirkels.
 - 2. Elke cirkel wordt gedefinieerd door het punt van de oorsprong en de diameter.
 - 3. Bereken of er tussen 2 opgegeven cirkels overlap is, dat is of een deel van een cirkel over de andere cirkel ligt.
 - 4. Bereken welke cirkels overlappen.
- b) Schrijf het codeskelet met javadoc documentatie van bovenstaande verantwoordelijkheden.

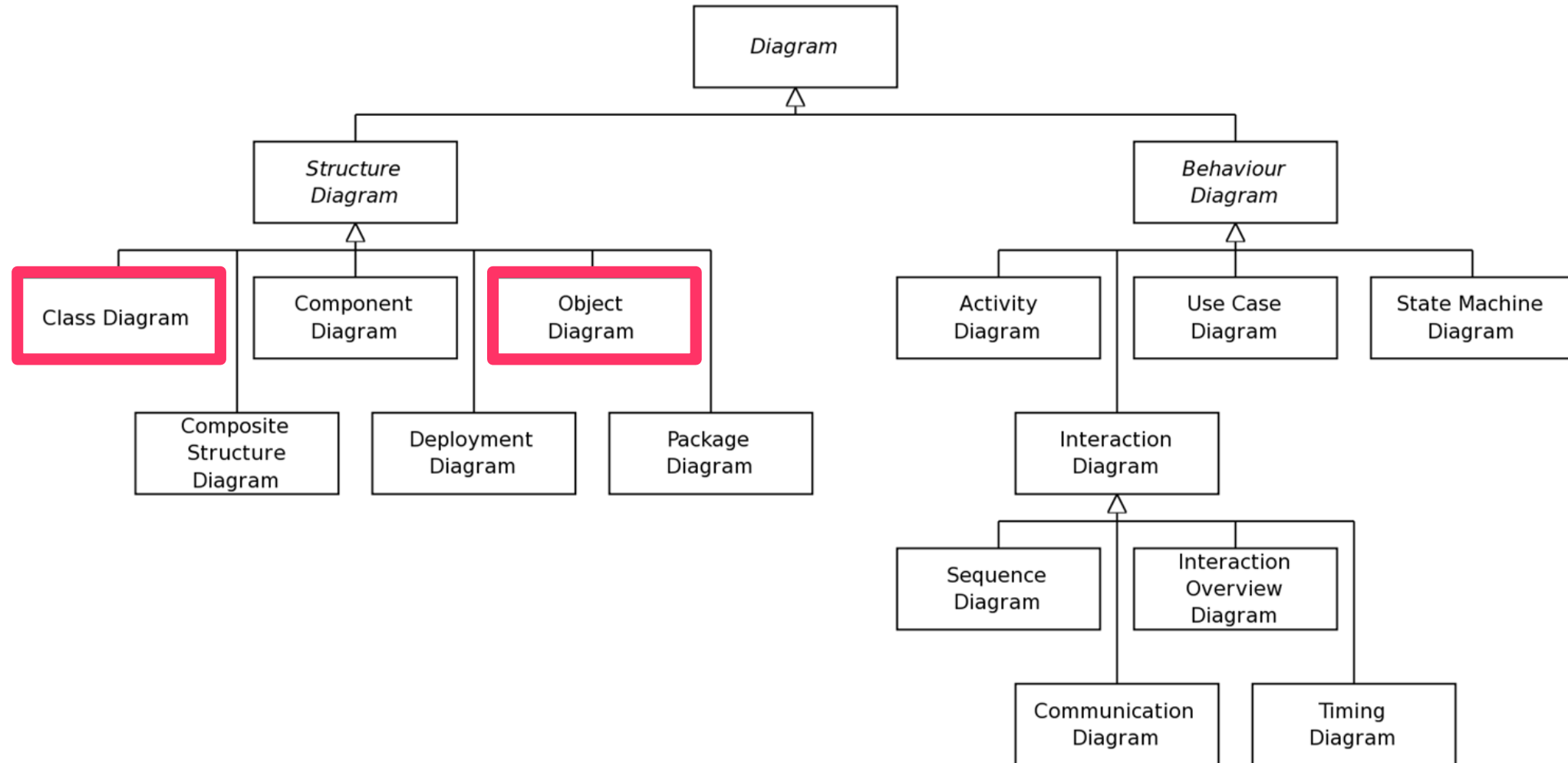


UML

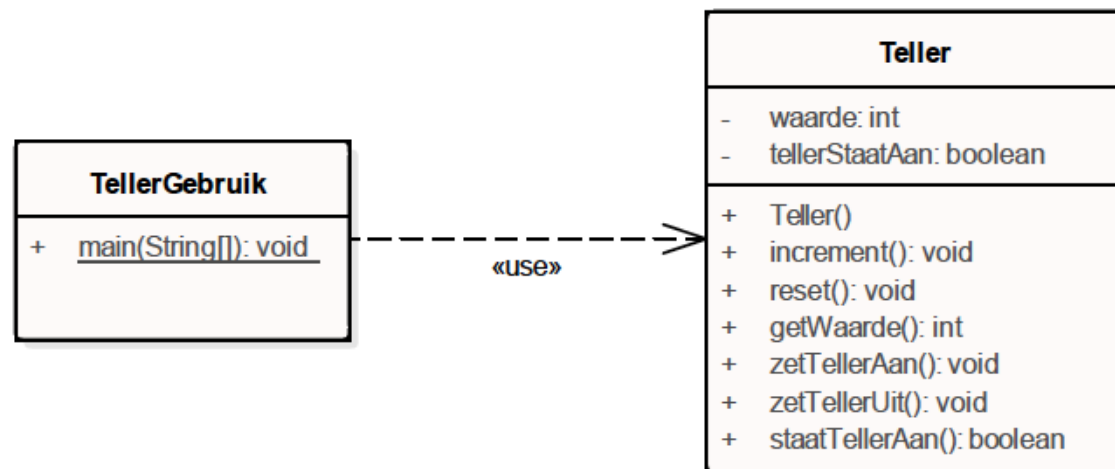
UML of Unified Modelling Language

- ▣ Set van symbolen en syntaxregels om software grafisch voor te stellen
- ▣ Tussen natuurlijke taal en code
- ▣ Voordelen
 - Gedachten ordenen
 - Discussiëren over de architectuur/ontwerp
 - Aftoetsen van volledigheid/kwaliteit: Is er aan alles gedacht? Wat ontbreekt?
 - Complexiteit reduceren en overzicht behouden
- ▣ Er bestaan tools om UML in code om te zetten en omgekeerd

Soorten UML-diagrammen



Het UML – klassendiagram van de Teller applicatie



Schema van een klasse



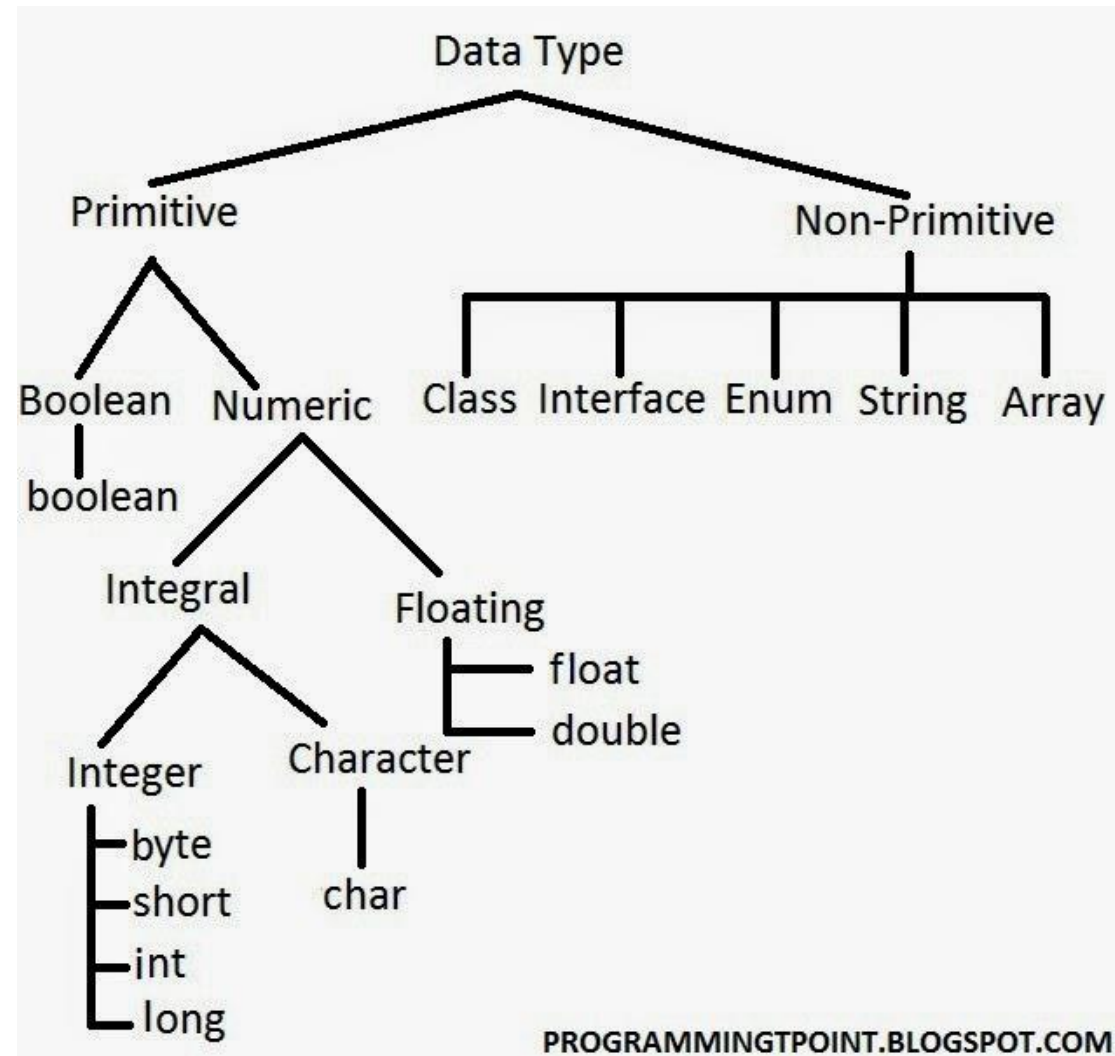
Attributen / Eigenschappen / Properties

Teller	
-	waarde: int
-	tellerStaatAan: boolean
+	Teller()
+	increment(): void
+	reset(): void
+	getWaarde(): int
+	zetTellerAan(): void
+	zetTellerUit(): void
+	staatTellerAan(): boolean

```
public class Teller {  
  
    private int waarde;  
    private boolean tellerStaatAan;  
}
```

Elk attribuut heeft 3 kenmerken:

1. **Zichtbaarheid**
 - voor private
 - + voor public
 - # voor protected
2. **Type**
 - Type of klasse van de variabele
3. **Naam**
 - begint met kleine letter heeft betekenis



Welke sleutelwoorden zijn er

▣ Voor visibility

- ▬ public
- ▬ private
- ▬ protected
- ▬ package

▣ static

- ▬ Property gedeeld door alle objecten in de klasse
- ▬ Property van de klasse, niet van het object

▣ final

- ▬ Voor constanten
- ▬ Kan enkel in de constructor of declaratie ingesteld worden



Methodes

Teller
- waarde: int - tellerStaatAan: boolean
+ Teller() + increment(): void + reset(): void + getWaarde(): int + zetTellerAan(): void + zetTellerUit(): void + staatTellerAan(): boolean

Elke methode heeft 3 kenmerken:

1. **Zichtbaarheid**
– voor private
+ voor public
voor protected

2. **Return type**
Type of klasse van de return-waarde

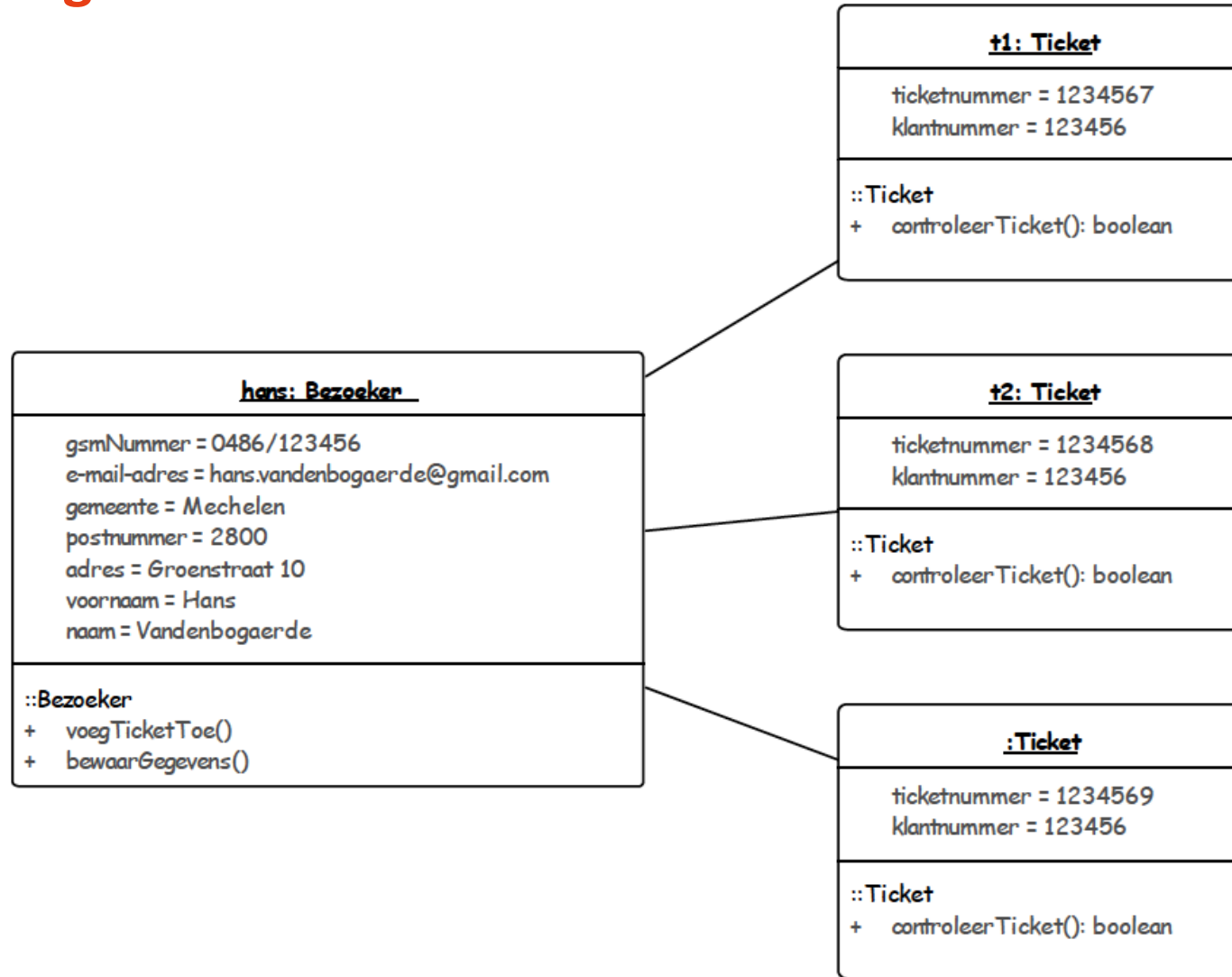
3. **Naam**
begint met kleine letter behalve de constructor

```
public Teller(){
    this.waarde = 0;
    this.tellerStaatAan = false;
}

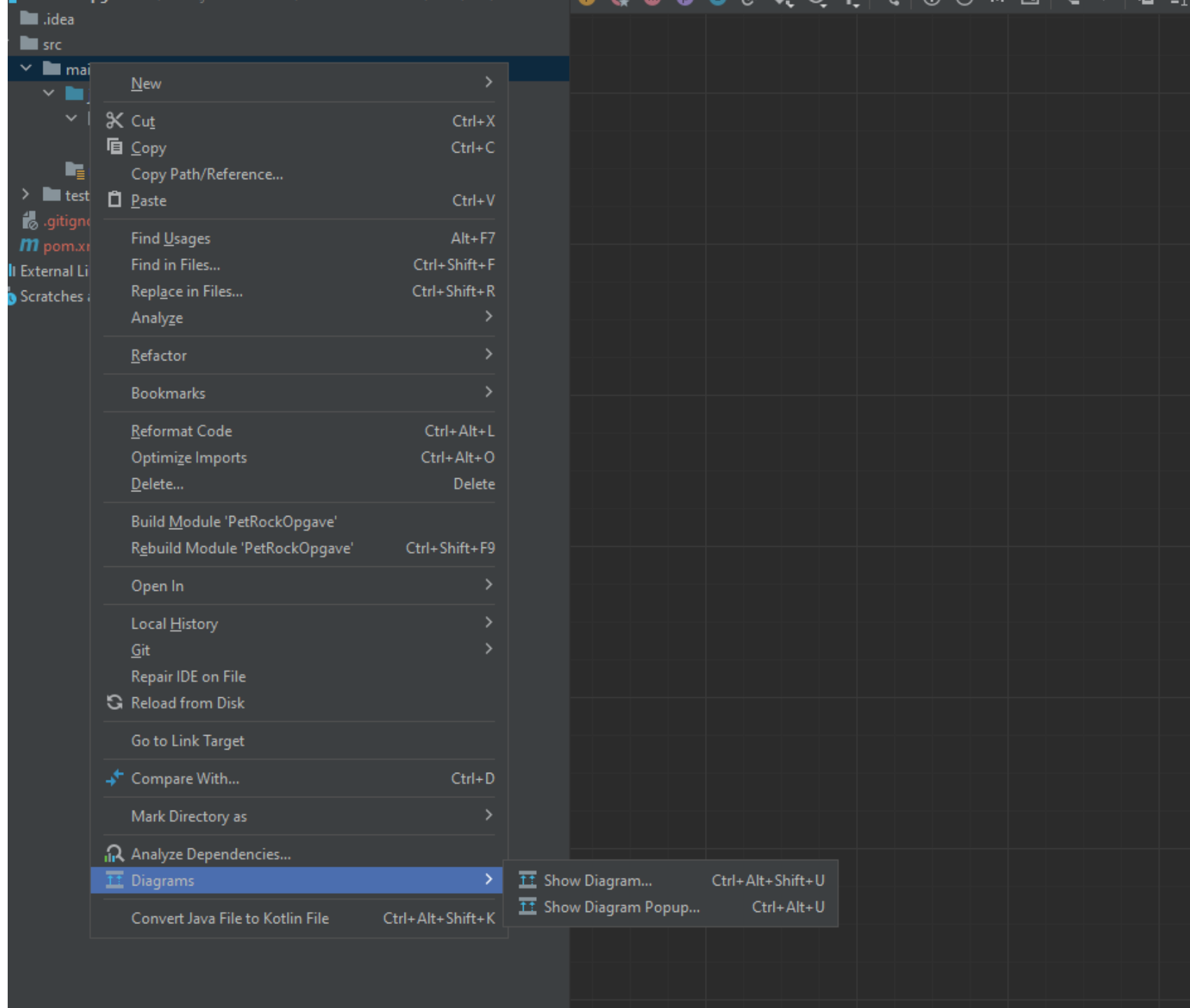
/**
 * Vermeerder waarde van Teller met 1.
 * Heeft geen effect als de teller uit staat.
 */
public void increment(){
    if (tellerStaatAan){
        this.waarde++;
    }
}
```

```
public int getWaarde(){
    if (tellerStaatAan) {
        return this.waarde;
    } else {
        return 0;
    }
}
```

UML – object diagram

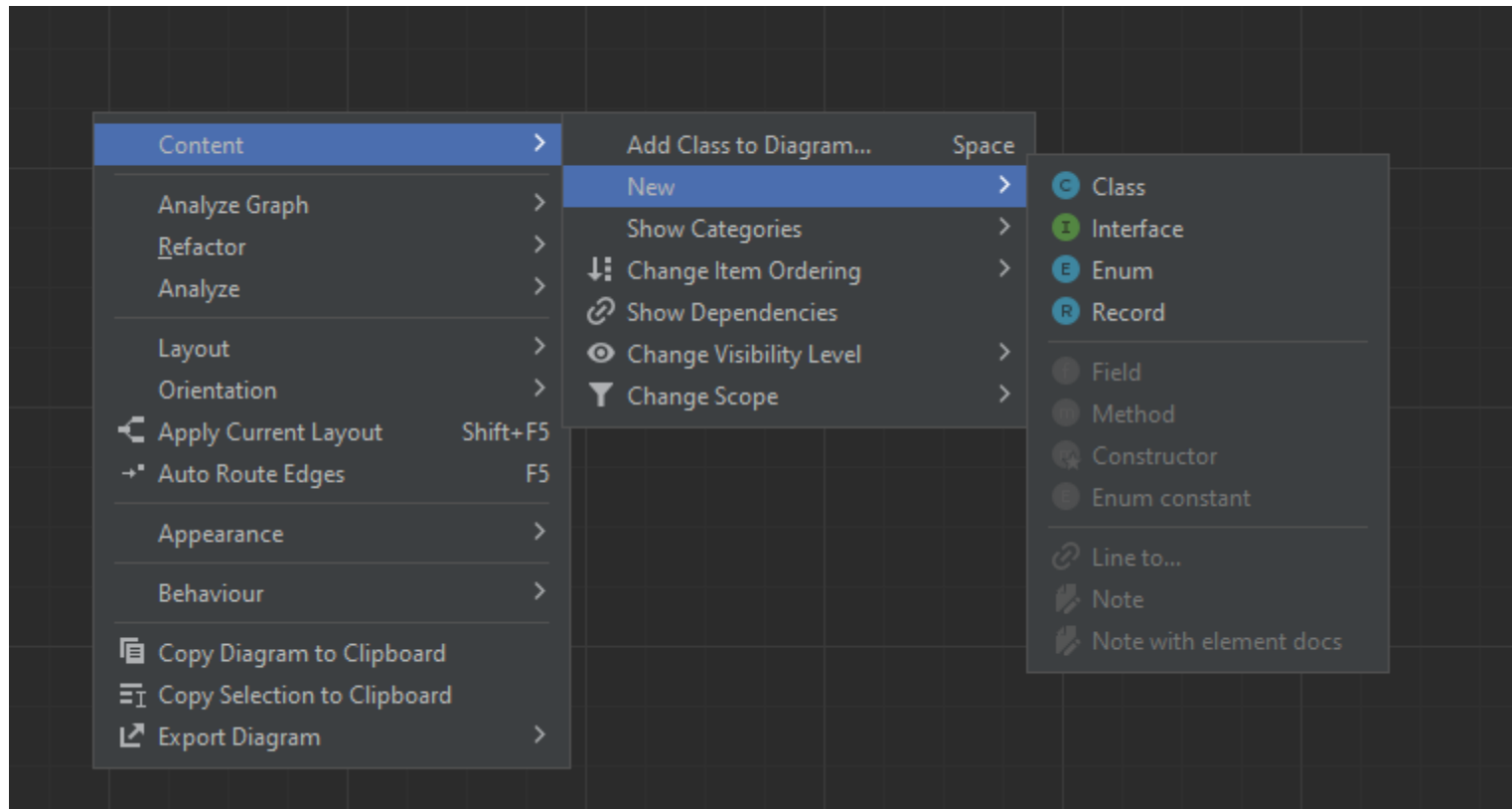


UML in IntelliJ



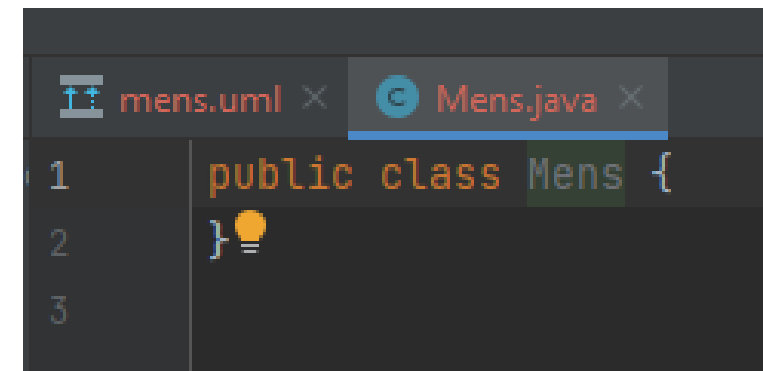
UML in IntelliJ

▣ Voeg een klasse Mens toe



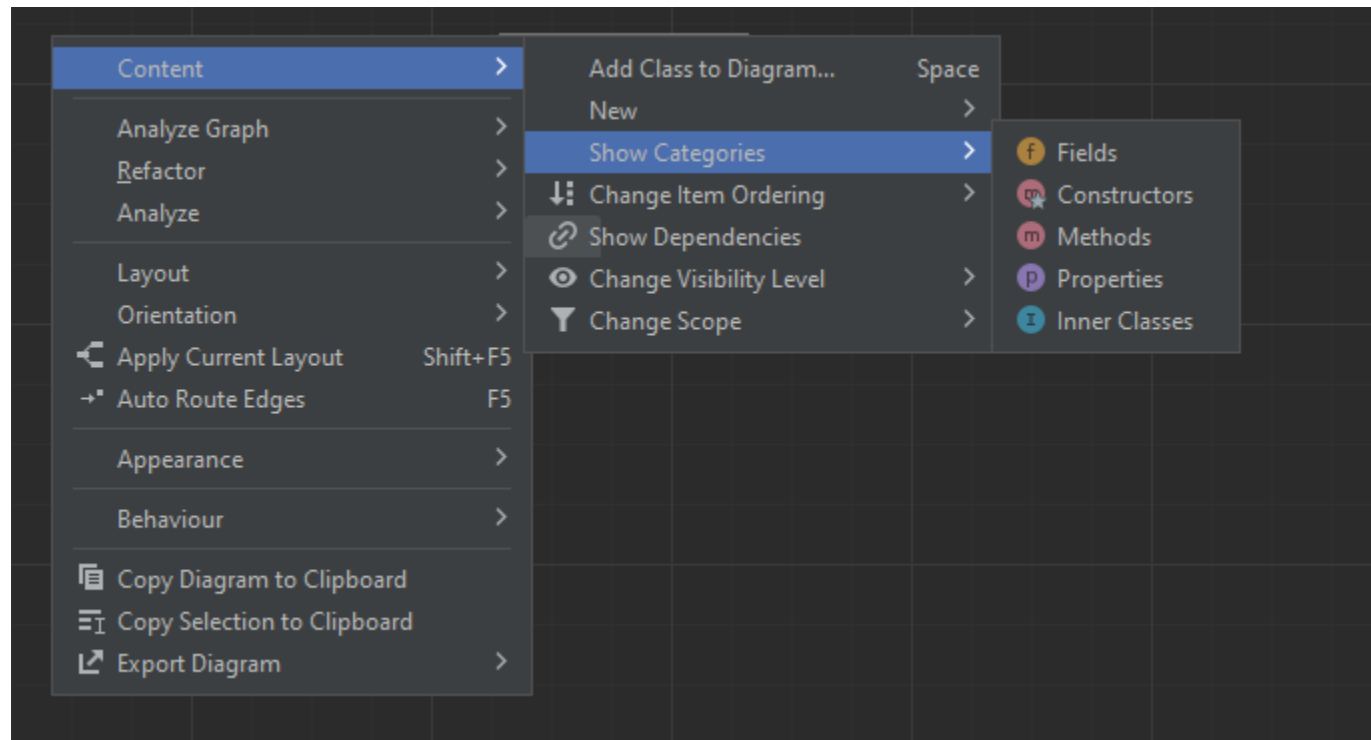
UML in IntelliJ

- ▣ Automatisch ook code-file toegevoegd

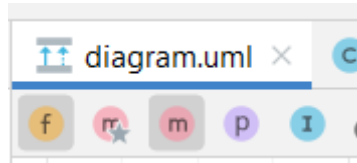


UML in IntelliJ

▣ Toon meer informatie

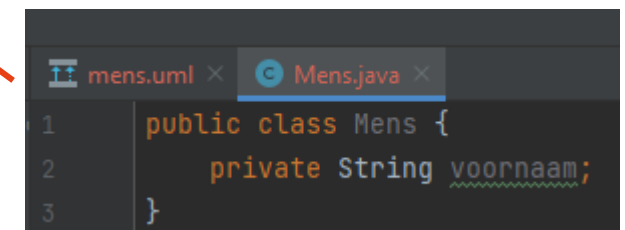
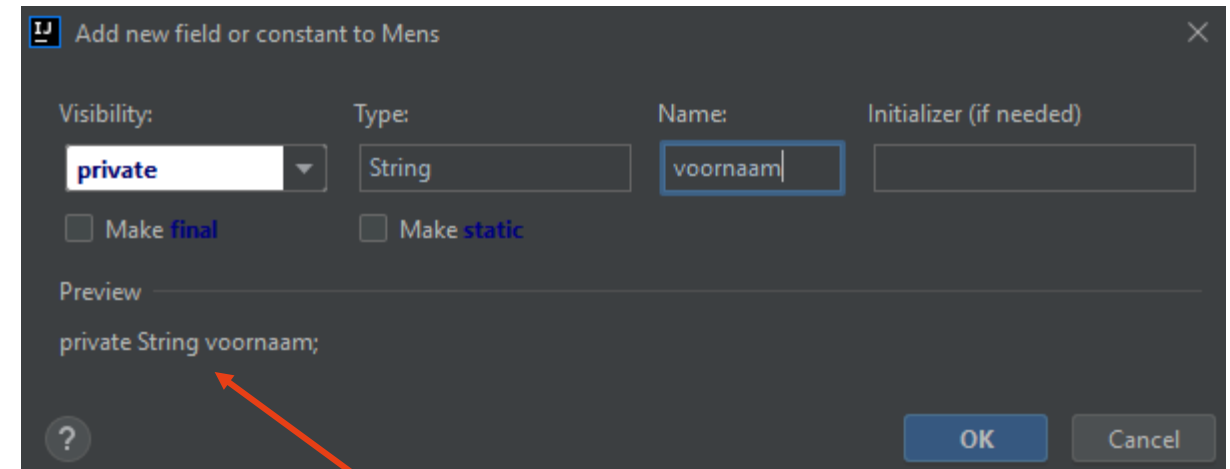
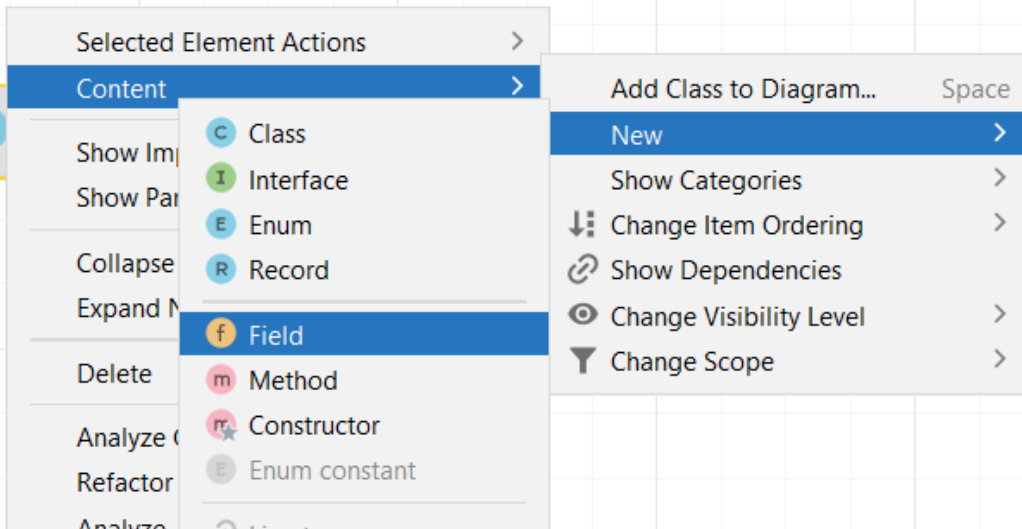


UML in IntelliJ



Toon fields en methods

- Voeg properties (fields) / methoden / constructors toe

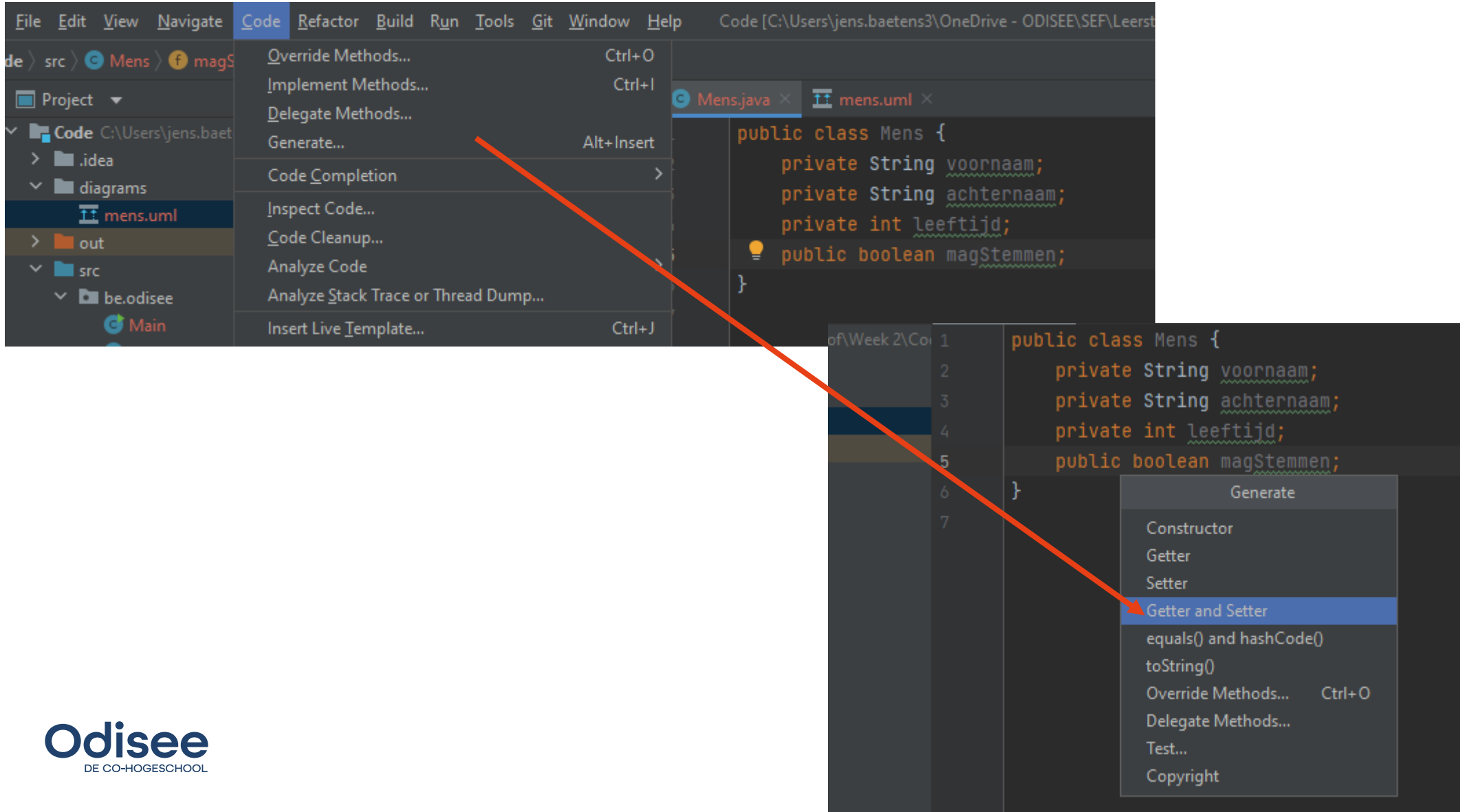




UML in IntelliJ

- ▣ Oefening: Voeg de volgende properties toe
 - achternaam: string – private
 - leeftijd: int – private
 - magStemmen: bool - public

Voeg getter en setters toe voor de private properties



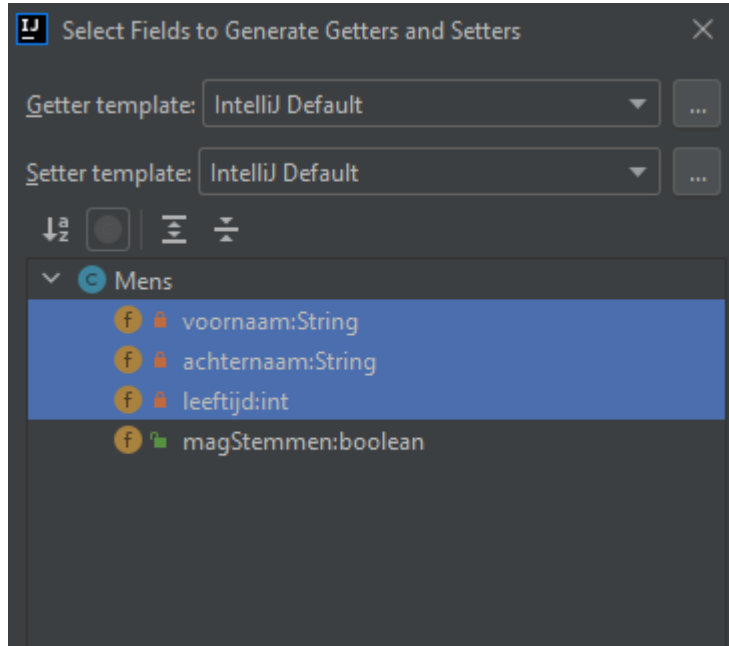
The screenshot shows the IntelliJ IDEA interface with the 'Code' menu open. The 'Generate' option is selected, which has opened a sub-menu. In this sub-menu, the 'Getter and Setter' option is highlighted. An orange arrow points from the 'Generate' button in the sub-menu to the 'Getter and Setter' option. The background shows the 'Mens.java' file with the following code:

```
public class Mens {  
    private String voornaam;  
    private String achternaam;  
    private int leeftijd;  
    public boolean magStemmen;  
}
```

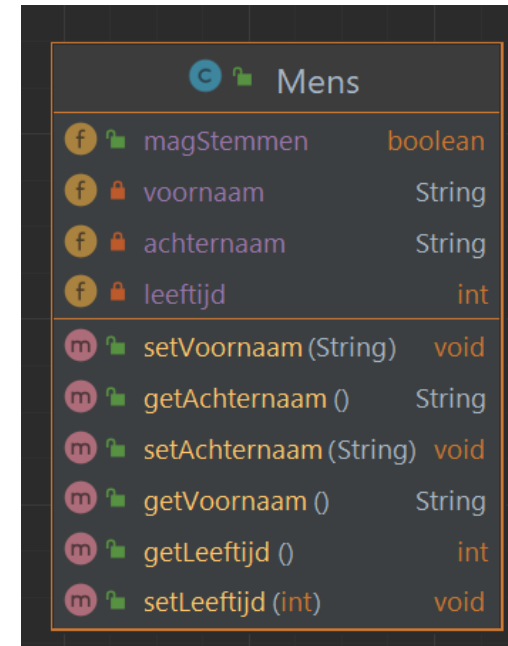
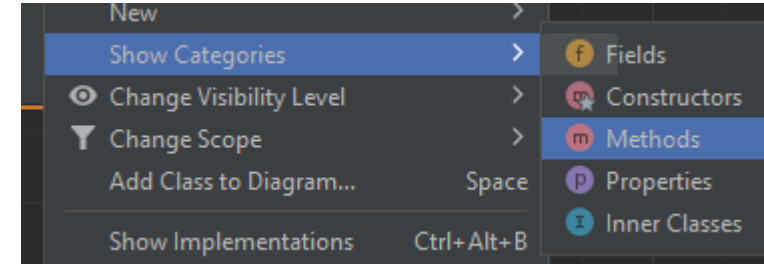
The sub-menu options are:

- Generate
- Constructor
- Getter
- Setter
- Getter and Setter
- equals() and hashCode()
- toString()
- Override Methods... Ctrl+O
- Delegate Methods...
- Test...
- Copyright

Voeg getter en setters toe voor de private properties



```
public String getVoornaam() {  
    return voornaam;  
}  
  
public void setVoornaam(String voornaam) {  
    this.voornaam = voornaam;  
}  
  
public String getAchternaam() {  
    return achternaam;  
}  
  
public void setAchternaam(String achternaam) {  
    this.achternaam = achternaam;  
}
```



■ Waarom moet de public property dit niet hebben?

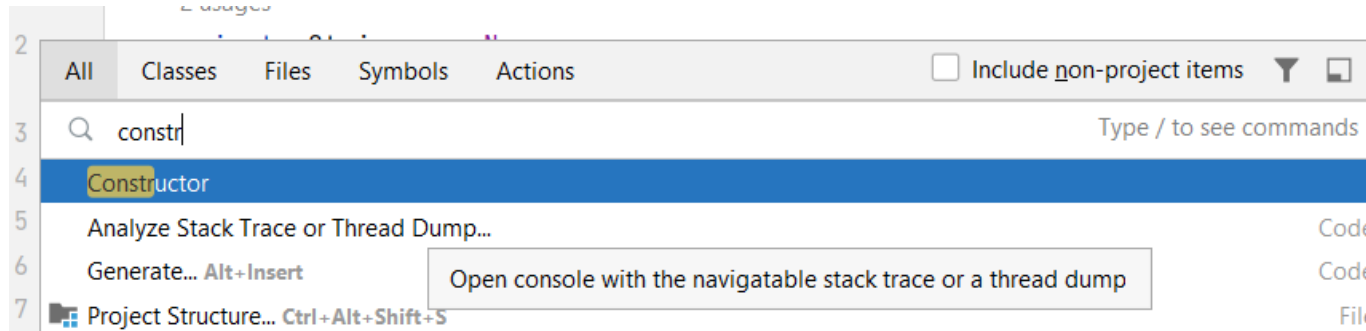


Oefening: Genereer een constructor

- ▣ Welke properties moeten gekozen worden bij het genereren van de constructor?

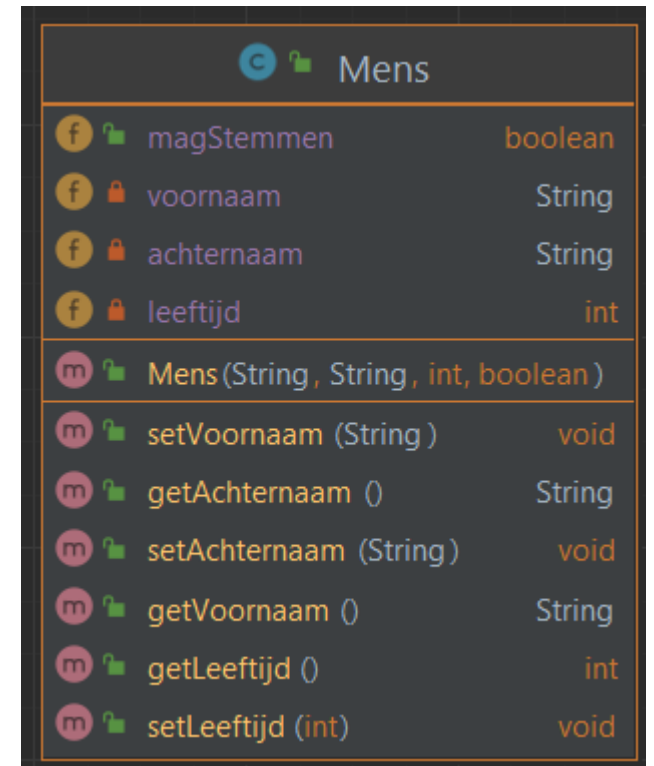
Oplossing: Genereer een constructor

- Gebruik de shortcut shift + shift voor het opzoeken van commands, code, ...



Resultaat

```
public Mens(String voornaam, String achternaam, int leeftijd, boolean magStemmen) {  
    this.voornaam = voornaam;  
    this.achternaam = achternaam;  
    this.leeftijd = leeftijd;  
    this.magStemmen = magStemmen;  
}
```



Oefening: Voeg methode lach toe

- ▣ Voeg via UML een methode lach toe
 - ▬ Return type is void
 - ▬ Geen parameters nodig
- ▣ Controleer in code of de methode is toegevoegd
- ▣ Implementeer de functie
 - ▬ Print naam van de gebruiker in console gevolgd door “Hahahahaha”

Oplossing: Voeg de methode lach toe

Create New Method

Visibility: [] Modifier: [none] Return type: void Name: lach

Parameters Exceptions

Nothing to show

Signature Preview

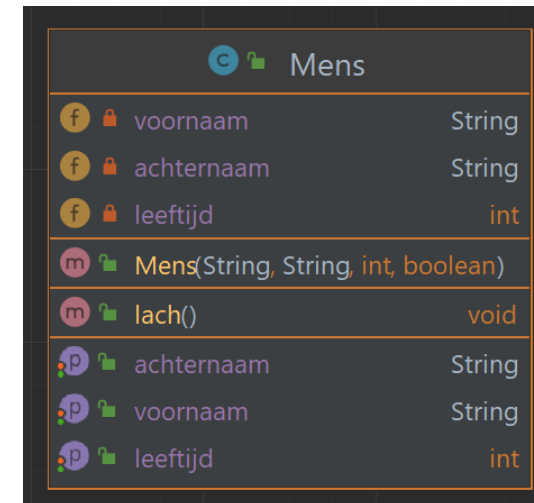
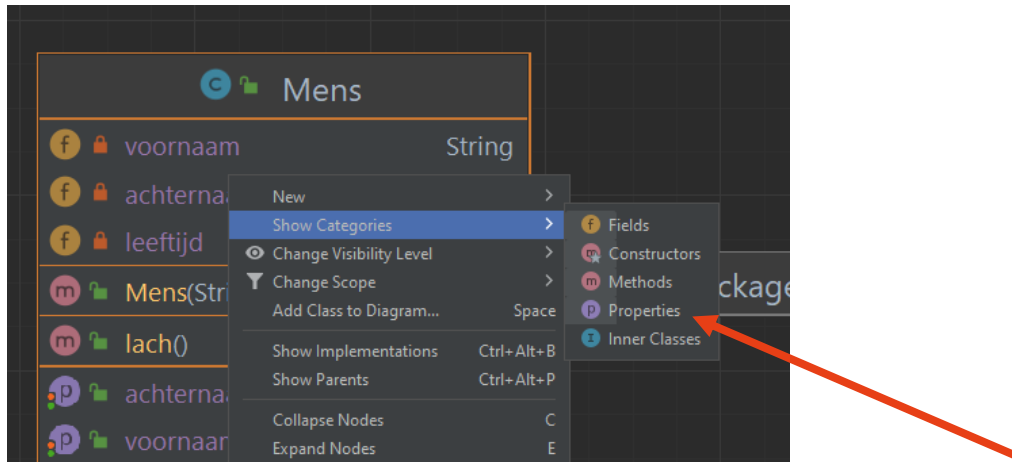
```
void lach()
```

Mens		
f	magStemmen	boolean
f	voornaam	String
f	achternaam	String
f	leeftijd	int
m	Mens (String, String, int, boolean)	
m	setVoornaam (String)	void
m	getAchternaam ()	String
m	lach()	void
m	setAchternaam (String)	void
m	getVoornaam ()	String
m	getLeeftijd ()	int
m	setLeeftijd (int)	void

```
public void lach() {  
    System.out.println(voornaam + " Hahahahahahaha");  
}
```

Overhead door getters en setters

- Getters en setters kunnen gegroepeerd worden door

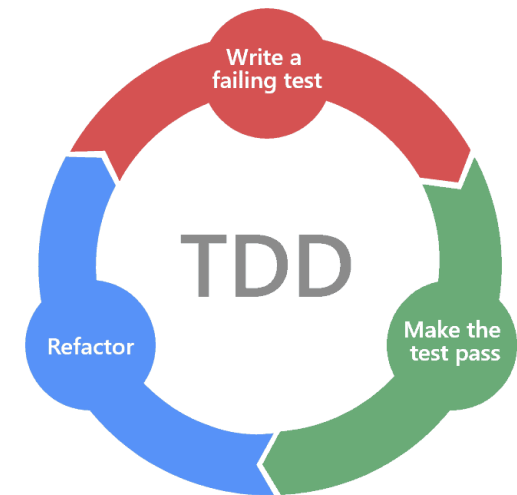




Test Driven Development

Test driven development

- ▣ Software ontwikkelingsproces sterk gesteund op schrijven van testen
- ▣ 3 fasen
 - Testing
 - Coding
 - Refactoring
- ▣ Kleine stappen met keer, volledig geteste code, ...



Testing framework in Java

▣ Testing framework

- ▬ Zoeken van testen
- ▬ Automatisch uitvoeren van testen
- ▬ Bijhouden welke slagen en welke falen (gewenste output / exception)

▣ Junit

- ▬ Voor C# is er NUnit

▣ Voor unit testing

- ▬ Test “units” van code
- ▬ Unit = zo klein mogelijk deel (vaak 1 methode)

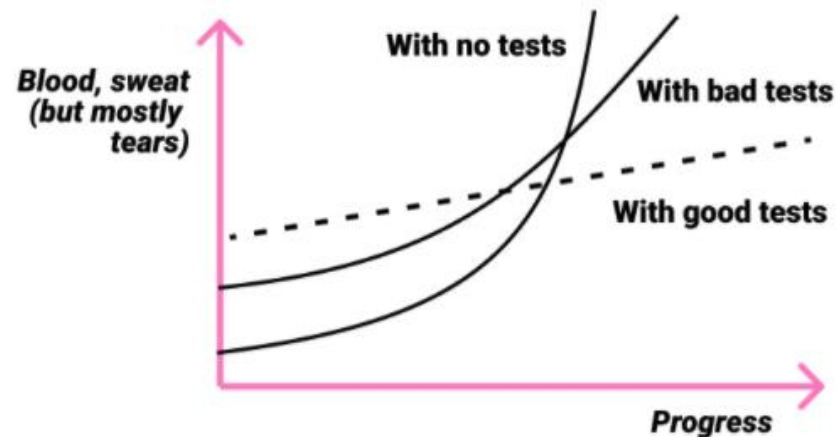
Soorten testen

▣ Acceptatietesten

- Testen of je geschreven applicatie voldoet aan de eisen van de klant

▣ Regressietesten

- Testen of na wijzigingen niet-aangepaste code nog steeds werkt

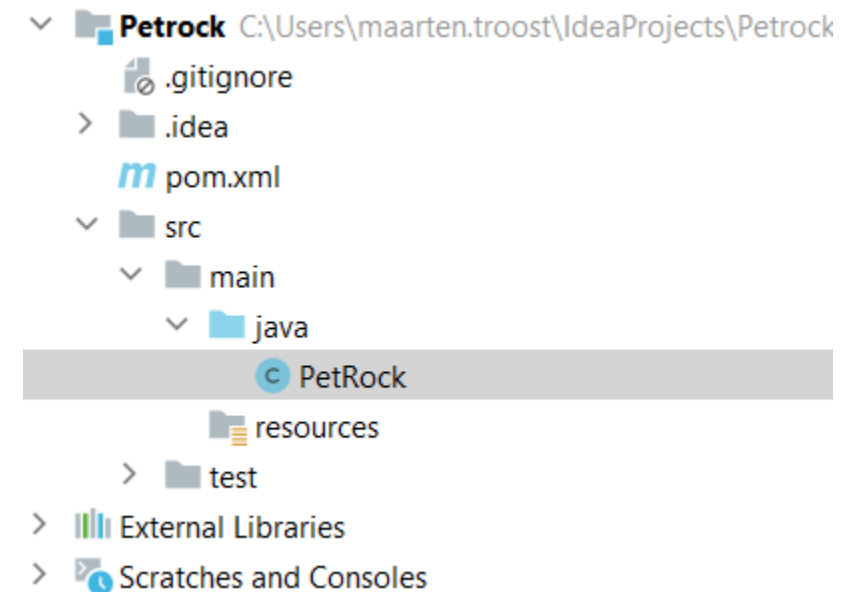


Demo PetRock

- Maak eenvoudige klasse aan

```
public class PetRock {  
  
    private String name;  
  
    public PetRock(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

- Zorg dat deze in de correcte directory staat
De src/main/java dir of een subdir hiervan



Demo Petrock

- ▣ Add Junit library to the pom.xml file

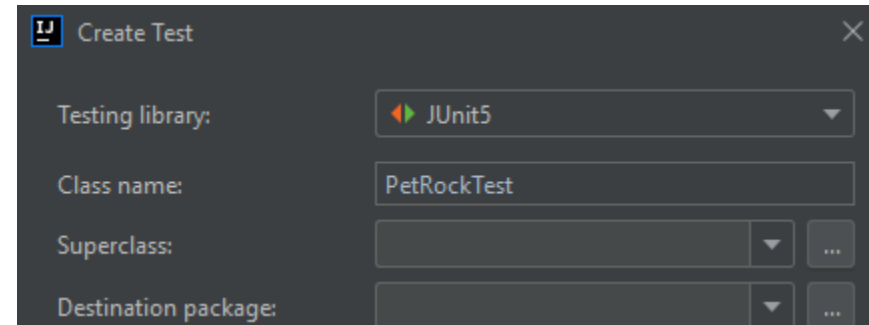
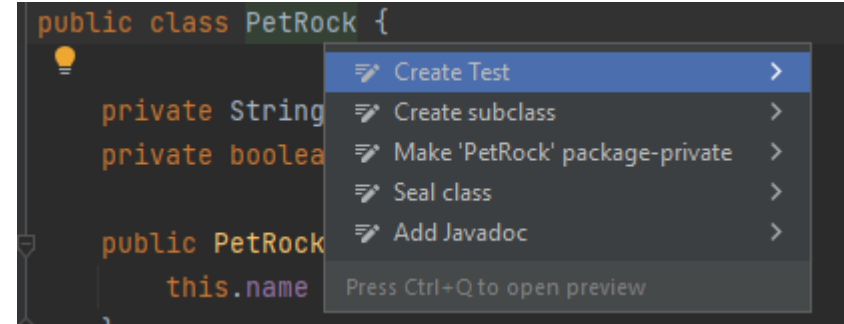
```
<dependencies>  
  <dependency>  
    <groupId>org.junit.jupiter</groupId>  
    <artifactId>junit-jupiter-api</artifactId>  
    <version>5.9.2</version>  
    <scope>test</scope>  
  </dependency>  
</dependencies>
```

- ▣ Zie ook <https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api>

Demo PetRock

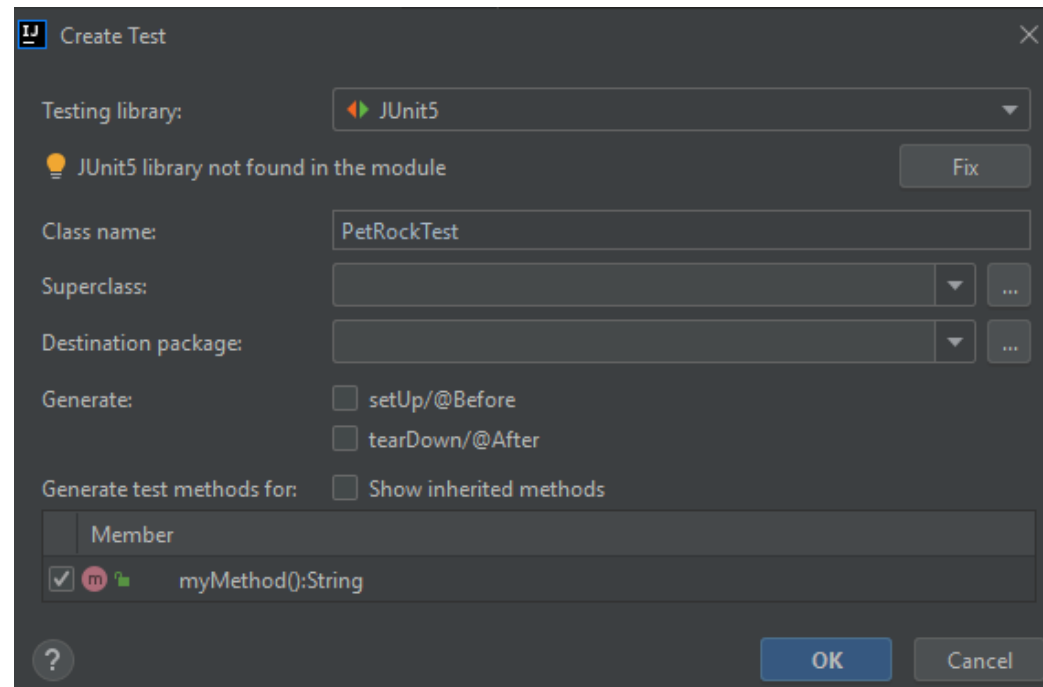
■ Maak een TestKlasse aan

- Selecteer naam klasse
- Alt + Enter -> Create Test
- OF: Generate -> Test



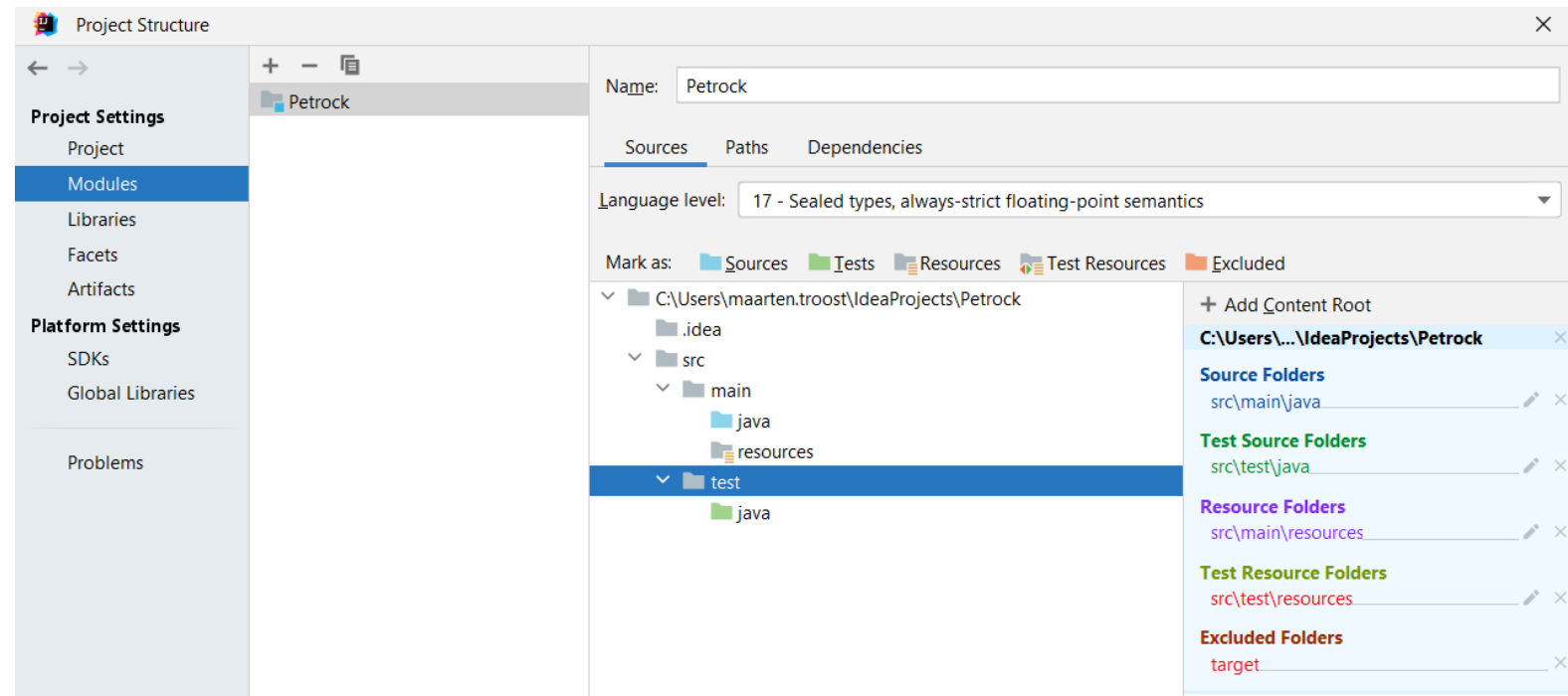
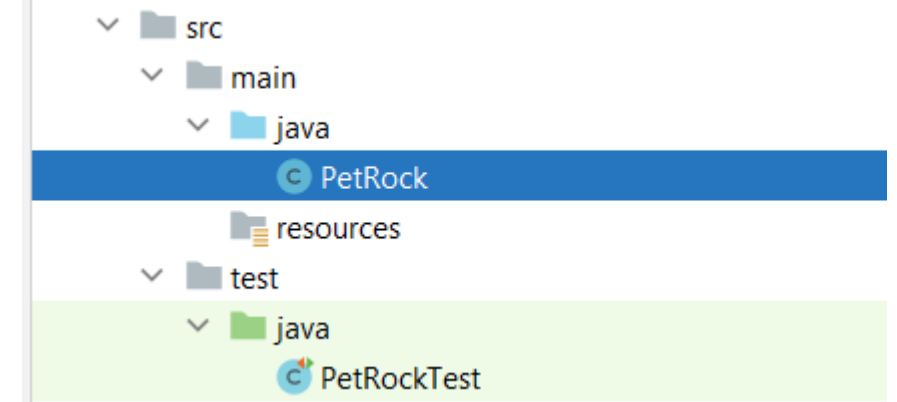
Demo PetRock: installatie JUnit

- ▣ Kies een naam = naam te testen klasse + Test
- ▣ Testingframework Junit5
- ▣ Selecteer voor welke methoden er een test moet zijn



Demo PetRock: TestKlasse

- Testen staan in de src/test dir of in de test directory. Deze kleurt groen. Indien niet pas je dit aan in de Project structure



Demo PetRock: TestKlasse

■ Annotatie @...Test

- Geeft aan dat de volgende methode een test is
- Indien hier een rode tag is, is er iets mis
 - Intellisense kan dit oplossen

```
import static org.junit.jupiter.api.Assertions.*;

class PetRockTest {

    @org.junit.jupiter.api.Test
    void getName() {
        PetRock rocky = new PetRock( name: "Rocky");

        assertEquals( expected: "Rocky", rocky.getName());
    }
}
```

- Testmethode maakt een object aan en test of de code doet wat we ervan verwachten
- Rechtsklik op de methode en kies run om de test uit te voeren



Programmeren via test driven development

- ▣ Voeg een test toe dat controleert of de PetRock gelukkig is of niet
 - ▬ Bij default is deze niet gelukkig

```
@Test
void isHappy_AtStart_ReturnsFalse(){
    PetRock rocky = new PetRock( name: "Rocky");

    assertFalse(rocky.isHappy());
}
```

- ▣ Deze test faalt omdat de isHappy() methode niet bestaat

```
C:\Users\jens.baetens3\OneDrive - ODISEE\SEF\L
java: cannot find symbol
  symbol:   method isHappy()
  location: variable rocky of type PetRock
```

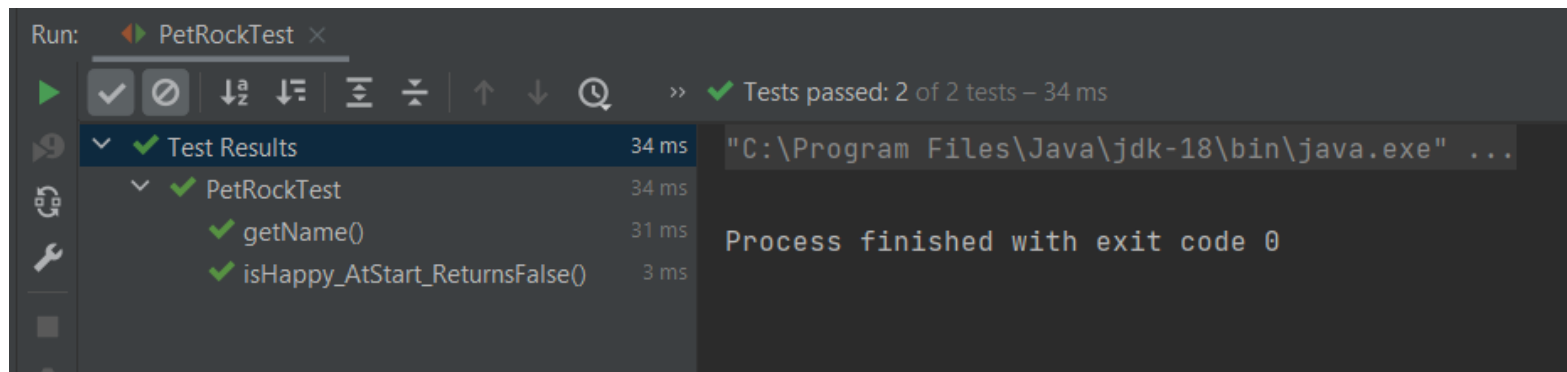
Test Results	34 ms	"C:\Program Files\Java\jdk-18\bin\jav
PetRockTest	34 ms	
✓ getName()	27 ms	
✗ isHappy_AtStart_ReturnsFalse()	7 ms	org.opentest4j.AssertionFailedError: Expected :false Actual :true <Click to see difference>

Programmeren via test driven development

- ▣ Nu dat test faalt, schrijf (minimale) code om test te doen slagen

```
public boolean isHappy(){  
    return false;  
}
```

- ▣ Alle testen slagen, schrijf opnieuw eerst test die faalt voor de nieuwe functionaliteit



Programmeren via test driven development

- ▣ Volgende functionaliteit die we willen toevoegen is dat het object gelukkig is nadat ermee gespeeld is
 - ▣ Maak hierbij gebruik van een `.play()` methode
- ▣ Schrijf opnieuw eerst de test voor er gecodeerd wordt

Programmeren via test driven development

```
@Test
void isHappy_AfterPlay_ReturnsTrue() {
    // Arrange or Given
    PetRock rocky = new PetRock( name: "Rocky");

    // Act or When
    rocky.play();

    // Assert or Then
    assertFalse(rocky.isHappy());
}
```

Waarom faalt deze test?

Programmeren via test driven development

- ▣ Vervolledig de code om de test te doen slagen

```
public class PetRock {  
  
    private String name;  
    private boolean happy = false;  
  
    public PetRock(String name) {  
        this.name = name;  
    }  
  
    public String getName() { return name; }  
  
    public boolean isHappy(){  
        return false;  
    }  
  
    public void play(){  
        happy = true;  
    }  
}
```

Best practices voor een goede test

- ▣ Leesbaarheid is heel belangrijk
 - ▬ Arrange – Act – Assert of Given – When – Then
- ▣ Act is maximaal 1 lijn, anders test je meerdere zaken
- ▣ In de assert moet je alle neveneffecten ook controleren en niet alleen de return waarde
 - ▬ Bvb als je een PetRock toevoegt aan je lijst van huisdieren

Best practices – Arrange Act Assert

```
@Test
void isHappy_AfterPlay_ReturnsTrue() {
    // Arrange or Given
    PetRock rocky = new PetRock( name: "Rocky");

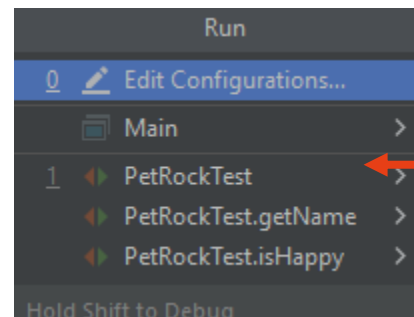
    // Act or When
    rocky.play();

    // Assert or Then
    assertFalse(rocky.isHappy());
}
```

JUnit in IntelliJ

▣ Belangrijke toetsen combinaties

- ▬ Alt + Enter om een testklasse aan te maken
 - Zorg ervoor via file -> project structure -> modules dat er een test directory is
- ▬ Alt + Shift + F10 om run config in te stellen
- ▬ Ctrl + Shift + F10 om geselecteerde config uit te voeren
 - Alle testen indien klasse geselecteerd
 - 1 test indien specifieke test geselecteerd



Alle testen uitvoeren
in deze klasse

Belangrijke annotaties

▣ @BeforeAll / @AfterAll

- Voor een gedeelde initialisatie van de testklasse
- Aanmaken data, opzetten connecties, downloaden informatie

▣ @BeforeEach / @AfterEach

- Voer initialisatie elke test opnieuw uit

OOP basics



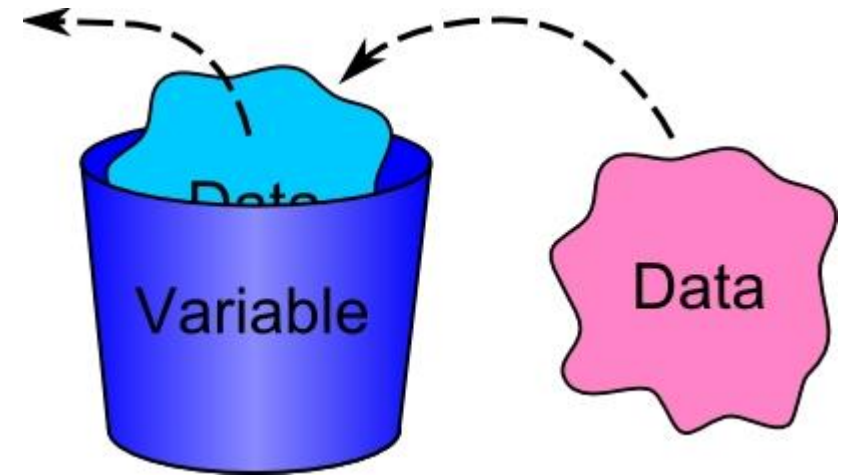
Maarten Troost – Jens Baetens

Variabele? Wasda?

In een programmeertaal gebruiken we variabelen om wisselende data in op te slaan.

```
int var=5;
```

Bij het uitvoeren van een programma, waar is die variabele op een fysieke computer?
Hoe worden meerdere variabelen van elkaar onderscheiden?



Variabele? Wasda?

- Een **variabele** is een **geheugenplaats** die via een naam wordt benaderd. De naam van de variabele is niets anders dan een **symbool** voor het adres in het geheugen.
 - ▬ Geheugenplaatsen = RAM, SSD, HDD, ...
 - ▬ Een plaats in het geheugen = een adres in het geheugen
vb 0x0003C4
 - ▬ Na compilatie geen namen meer maar enkel adressen

Variabelen in java

- ▣ Een waarde toekennen aan een variabele betekent dat we gegevens stoppen in het geheugen waarnaar de variabele verwijst. In Java gebruiken we hiervoor het “=”-teken ook de assignment operator genoemd.

Vb. `int var = 6;`

- ▣ Waarden vergelijken doen we met “==” (en “!=”), comparison operators voor primitieve types, de `.equals` methode voor objecten.

```
if(var == 6) {  
    String str=new String("meh");  
    if(str.equals("beeeh")) {
```

```
@Override  
public boolean equals(Object obj) {
```

Operatoren – prioriteit en associativiteit

Precedence	Operator	Associativity	Operator
1	++	Right	Pre/post increment
	--	Right	Pre/post decrement
	+, -	Right	Unary plus or minus
	~	Right	Bitwise complement
	!	Right	Logical complement
	(cast)	Right	Cast
2	*, /, and %	Left	Multiplication, division, and modulus
3	+ and -	Left	Addition and subtraction
	+	Left	String concatenation
4	<<	Left	Left shift
	>>	Left	Right shift and sign fill
	>>>	Left	Right shift and zero fill
5	<, <=, >, >=	Left	Logical
	InstanceOf	Left	Type comparison
6	== and !=	Left	Equality and inequality
7	&	Left	Bitwise and Boolean AND
8	^	Left	Bitwise and Boolean XOR
9		Left	Bitwise and Boolean OR

Operatoren – prioriteit en associativiteit

Precedence	Operator	Associativity	Operator
10	&&	Left	Boolean AND
11		Left	Boolean OR
12	?:	Right	Conditional
13	=	Right	Assignment
	+=, -=, *=, /=, and %=	Right	Compound

Bron: Reese - OCA, Java SE7 Programmer Study Guide - Packt 2012

For operators, associativity means that when the same operator appears in a row, then which operator occurrence we apply first. In the following, let `q` be the operator

`a q b q c`

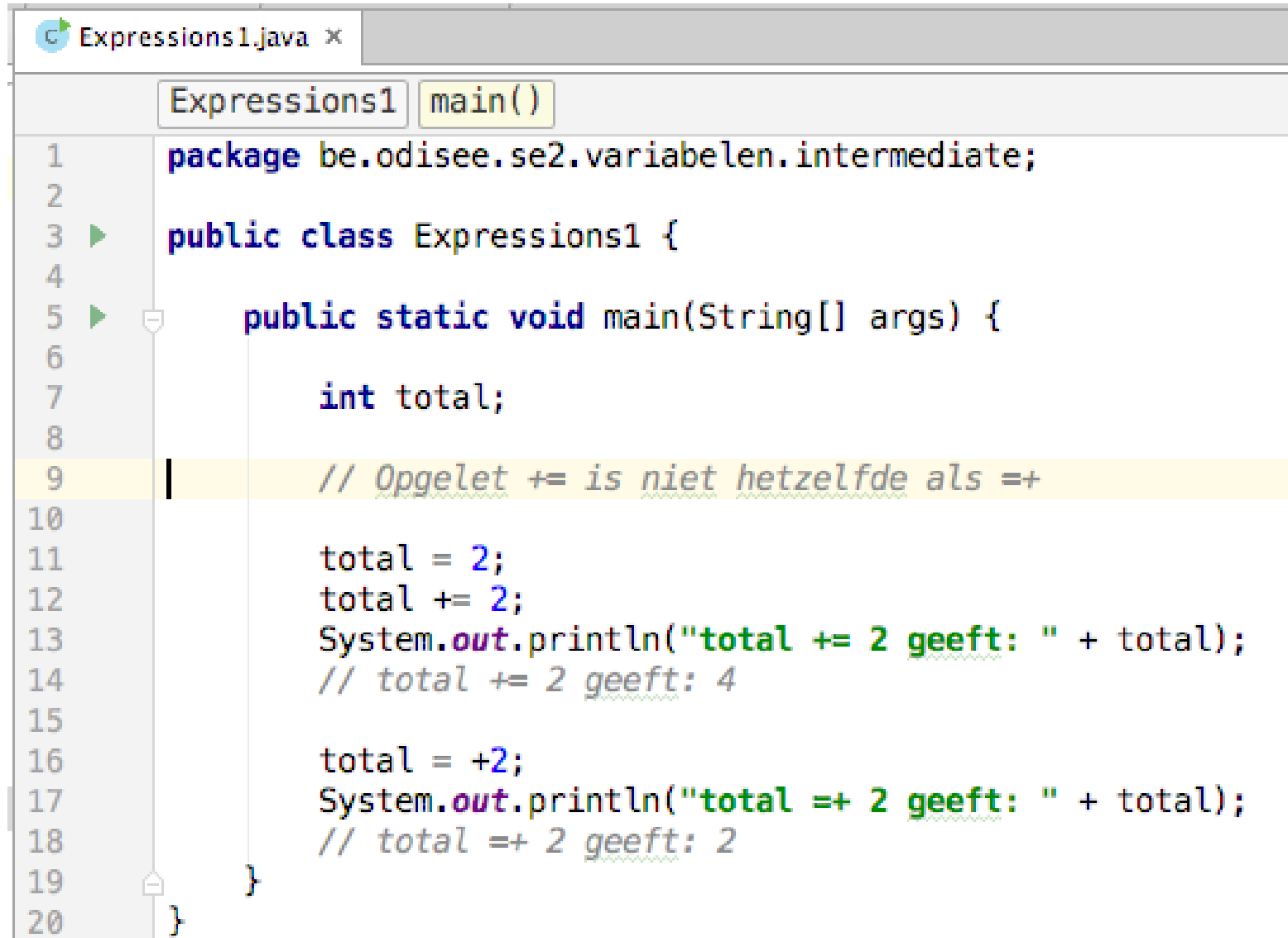
If `q` is left associative, then it evaluates as

`(a q b) q c`

And if it is right associative, then it evaluates as

`a q (b q c)`

Operatoren – oefening: leg uit wat de uitkomst is

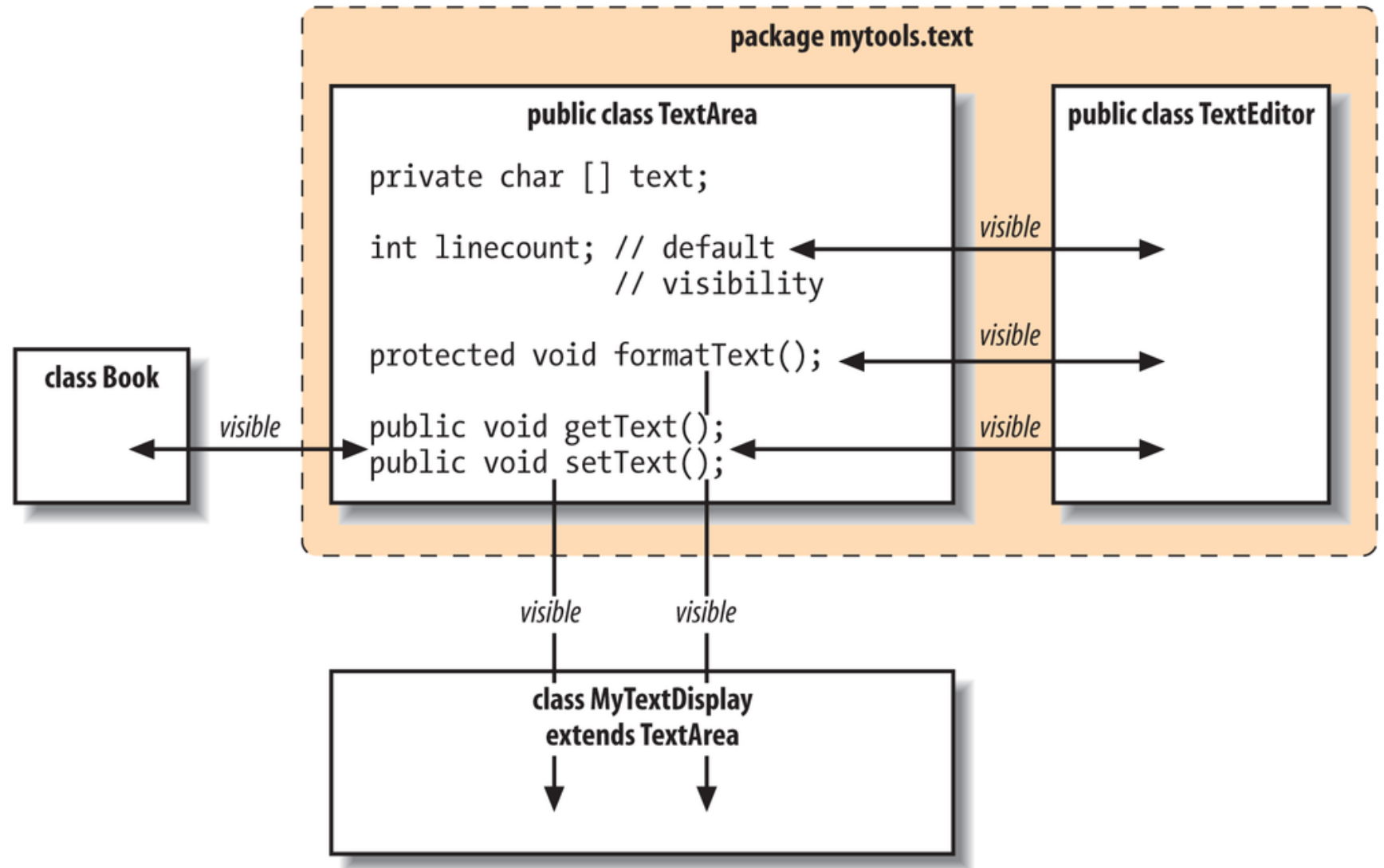


```
Expressions1.java x
Expressions1 main()
1 package be.odisee.se2.variabelen.intermediate;
2
3 public class Expressions1 {
4
5     public static void main(String[] args) {
6
7         int total;
8
9         // Opgelet += is niet hetzelfde als +=
10
11         total = 2;
12         total += 2;
13         System.out.println("total += 2 geeft: " + total);
14         // total += 2 geeft: 4
15
16         total = +2;
17         System.out.println("total += 2 geeft: " + total);
18         // total += 2 geeft: 2
19     }
20 }
```

Visibility

Access modifiers

- public
overal zichtbaar
- private
eigen class only
- protected
eigen class
+ derived classes
+ eigen package
- package (=default)
eigen class
+ eigen package



Bron: O'REILLY

Visibility

■ Waarom visibility?

- Information hiding: beschermen van gedrag (methods) en data (at run time) tegen wijzigingen. Enkel invloed van beperkte bron toestaan = voorspelbaarder gedrag
- Code locality: Alle code over X moet bij elkaar staan

■ Wanneer welke access modifier?

- private
Zo veel mogelijk
- protected
Indien subclassing en relevant voor extenties
- package (=default)
Indien samenwerking met andere classes (aan hetzelfde onderwerp = package)
- public
Enkel indien nodig of constanten

To be static or not to be static

▣ static

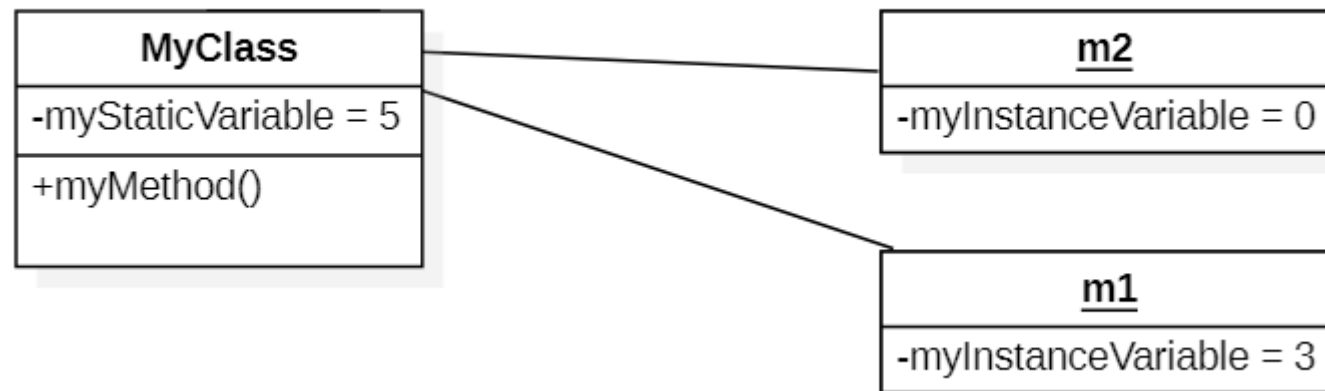
- ▬ Variabele gedeeld door alle objecten in de klasse
- ▬ Variabele van de klasse, niet van het object

private static int *myStaticVariable*;

MyClass.*myStaticVariable*++;

private int *myInstanceVariable*;

MyClass m1=new MyClass();
m1.*myInstanceVariable*++;





```
/** Stelt een fysieke teller voor welke aan de deur staat en telt hoeveel personen binnen en buitengaan.  
Bedient door mensen. */
```

```
public class ToegangsControleStaticExample {
```

```
/** Deze variable onthoudt hoeveel fysieke tellers (= instances van deze class) er gemaakt zijn. */
```

```
private static int aantalTellers=0;
```

```
ToegangsControleStaticExample() {
```

```
aantalTellers++; //Ctor: nieuwe teller in gebruik
```

```
}
```

```
public static void verwijderTeller() {
```

```
aantalTellers--; //teller wordt uit gebruik gehaald
```

```
}
```

```
public static boolean zijnErNogActieveTellers() {
```

```
return aantalTellers > 0; //zijn er nog tellers in gebruik
```

```
}
```

```
/** onthoudt het aantal mensen dat door 1 deur zijn binnengetreden (en niet zijn buitengetreden)
```

```
* 1 aantal per fysieke teller */
```

```
private int aantal = 0;
```


To be static or not to be static

■ Wanneer static gebruiken?

▬ Variabelen

- Constanten (naam constante in HOOFDLETTERS)
vb `Integer.SIZE = 32;`
- Hoeveel objecten zijn er van mijn class? Nodig als er maar 1 instance mag zijn.
vb maar 1 connectie met de database

▬ Methods

- Als een method niet afhankelijk is van een instance variabele
vb `Integer.getInteger(String name)` en `Integer.getInteger(String name, int defaultValue)`

Constanten

▣ final variable

- ▬ Laat niet toe de waarde te wijzigen na initialisatie
- ▬ Kan in de constructor ingesteld worden, niet in een method

```
class myFinalExampleClass {  
    /* Kan ingesteld worden bij declaratie of in constructor  
       NIET IN BEIDE TEGELIJK! */  
    private final int myVariable=5;  
    myFinalExampleClass(int startWaarde) {  
        myVariable=startWaarde;  
    }  
}
```

Constanten

▣ Wanneer final gebruiken?

- ▬ Als een waarde niet meer gewijzigd mag worden
- ▬ Constanten
- ▬ Zo veel mogelijk: het vermijdt onbedoelde wijzigingen

```
public int machtsverheffing(int grondtal, final int exponent) {  
    int result=1;  
    for(int i=exponent; i>0 ; i--) {  
        grondtal*=grondtal; //foute code maar geen syntax error  
        exponent--; //foute code en syntax fout  
    }  
    return result;  
}
```

final object != final class variables

- Opgelet! Het final keyword laat niet toe om de waarde van de variabele te wijzigen naar een *ander* object.

Wel toegelaten is het wijzigen van de (niet final) variabelen van het final object.

```
private class MySubClass {  
    1 usage  
    public int var=5;  
}  
2 usages  
final MySubClass sub=new MySubClass();  
public void someMethod() {  
    sub=new MySubClass(); //mag niet want final  
    sub.var=6; //mag wel want var is niet final  
}
```

Oefening equality

```
@Override  
public boolean equals(Object obj) {
```

- Maak een class aan met minstens 2 instance variabelen (type vrij te kiezen). Implementeer de equals method.
Test of de == operator en equals de te verwachten waarden returnen.
Gebruik TDD

```
final Account ac1= new Account(35,"Draak");  
final Account ac2= new Account(35,"Draak");  
final Account ac3= new Account(666,"Boss");  
System.out.printf("Zijn deze account identiek? %b",ac1.equals(ac2));  
System.out.printf("Zijn deze account identiek? %b",ac1.equals(ac3));  
final Account ac4=ac1;  
System.out.printf("Zijn het dezelfde objecten? %b",ac1==ac2);  
System.out.printf("Zijn het dezelfde objecten? %b",ac1==ac4);
```

Defining equality opoassing test class

```
@Test void equals_identical_returnsTrue() {  
    //Arrange  
    Account a1=new Account(3,"Goblin");  
    Account a2=new Account(3,"Goblin");  
    //Act  
    boolean result=a1.equals(a2);  
    //Assert  
    assertTrue(result);  
}
```

```
@Test void  
equals_notIdenticalLevel_returnsFalse() {  
    Account a1=new Account(3,"Goblin");  
    Account a2=new Account(4,"Goblin");  
    assertFalse(a1.equals(a2));  
}
```

```
@Test void  
equals_notIdenticalCreature_returnsFalse() {  
    Account a1=new Account(3,"Goblin");  
    Account a2=new Account(3,"Roc");  
    assertFalse(a1.equals(a2));  
}
```

Defining equality overriding implementation code

```
public class Account {  
    private final int level;  
    private final String creature;  
    public Account(int level, String creature) {  
        this.level = level;  
        this.creature = creature;  
    }  
    @Override  
    public boolean equals(Object obj) {  
        Account acc=(Account)obj;  
        return acc.level==this.level && acc.creature.equals(this.creature);  
    }  
}
```

Enum(eratie)

As an example, which is better?

```
/** Counts number of foobangs.
 * @param type Type of foobangs to count. Can be 1=green foobangs,
 * 2=wrinkled foobangs, 3=sweet foobangs, 0=all types.
 * @return number of foobangs of type
 */
public int countFoobangs(int type)
```

versus

```
/** Types of foobangs. */
public enum FB_TYPE {
    GREEN, WRINKLED, SWEET,
    /** special type for all types combined */
    ALL;
}

/** Counts number of foobangs.
 * @param type Type of foobangs to count
 * @return number of foobangs of type
 */
public int countFoobangs(FB_TYPE type)
```


Enum(eratie)

A method call like:

```
int sweetFoobangCount = countFoobangs(3);
```

then becomes:

```
int sweetFoobangCount = countFoobangs(FB_TYPE.SWEET);
```

In the second example, it's immediately clear which types are allowed, docs and implementation cannot go out of sync, and the compiler can enforce this. Also, an invalid call like

```
int sweetFoobangCount = countFoobangs(99);
```

is no longer possible.

Bron: <http://stackoverflow.com/questions/4709175/what-are-enums-and-why-are-they-useful>

Enum(eratie)

- ▣ Enum's zijn zelf te ontwerpen "datatypes" welke een opsomming bevatten van de mogelijke waarden.
- ▣ Is een soort van lite class. Kan ook methods en constructors bevatten.

```
private enum Richtingen {NOORD, OOST, WEST, ZUID };  
private void useIt() {  
    Richtingen richting=Richtingen.OOST;  
    switch(richting) {  
        case NOORD -> System.out.println("Onward!");  
        case OOST  -> System.out.println("Take a right");  
        case ZUID  -> System.out.println("Please make a U-turn.");  
        case WEST  -> System.out.println("Neen, de andere rechts");  
    }  
}
```

Enum(eratie)

```
private enum Richtingen {  
    NOORD(0), OOST(90), ZUID(180), WEST(270);  
    private final int graden;  
    Richtingen(int graden) {  
        this.graden=graden;  
    }  
    public int getGraden() { return graden; }  
};
```

```
public static void main(String[] args) {  
    Richtingen mijnRichting;  
    mijnRichting=Richtingen.WEST;  
    System.out.println("De richting is "+mijnRichting.getGraden()+" graden.");  
}
```

Enum – Wanneer en hoe te gebruiken?

- ▣ Indien een variabele maar enkele toegelaten waarden heeft
- ▣ Vaak gedefinieerd in een class omdat de enum verbonden is aan de implementatie ervan.
- ▣ public indien gebruikt door een publieke functie als return of param waarde
- ▣ Naam enum begint met Hoofdletter. Namen van de waarden in HOOFDLETTERS.

```
private enum Richtingen {NOORD, OOST, WEST, ZUID };
```

Enum – Oefening opgave

▣ Basic:

- ▬ Stel de dagen van de week voor: maandag .. zondag
- ▬ Maak een functie nextDay welke de volgende dag teruggeeft.
- ▬ Denk na over access modifiers, final, ...

▣ Uitdaging:

- ▬ Maak ook een prevDay
- ▬ En een getDayOfWeek welke een int X returned voor de Xde dag van de week
- ▬ Vermijd code duplicatie

Enum – Oefening oplossing basic

```
/** Represents the days of the week of the Gregorian calendar */
private enum WeekDays {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY ;

    /** @return the next day of the week */
    public WeekDays nextDay() {
        switch(this) {
            case MONDAY: return TUESDAY;
            case TUESDAY: return WEDNESDAY;
            case WEDNESDAY: return THURSDAY;
            case THURSDAY: return FRIDAY;
            case FRIDAY: return SATURDAY;
            case SATURDAY: return SUNDAY;
            default: return MONDAY;
        }
    }
}
```

Enum – Oefening oplossing uitdaging

```
private enum WeekDay {
    MONDAY(1),
    TUESDAY(2),
    WEDNESDAY(3),
    THURSDAY(4),
    FRIDAY(5),
    SATURDAY(6),
    SUNDAY(7)
;
    private final int dayOfWeek;
    /** The number of the day starting with
    monday = 1 */
    public int getDayOfWeek() {
        return dayOfWeek;
    }
}
```

```
WeekDay(final int dayOfWeek) {
    this.dayOfWeek=dayOfWeek;
}
/** translates number into to WeekDays */
private static WeekDay toWeekDay(final int day)
{
    switch(day) {
        case 1: return MONDAY;
        case 2: return TUESDAY;
        case 3: return WEDNESDAY;
        case 4: return THURSDAY;
        case 5: return FRIDAY;
        case 6: return SATURDAY;
        default: return SUNDAY;
    }
}
```

Enum – Oefening oplossing uitdaging

```
/**
 * calculates the next day of the week
 * @return the next day of the week
 */
public WeekDay nextDay() {
    return toWeekDay(dayOfWeek%7+1);
}

/**
 * calculates the previous day of the week
 * @return the previous day of the week
 */
public WeekDay prevDay() {
    return toWeekDay(((dayOfWeek-2)%7)+1);
}
```

```
public static void main(String[] args) {
    final WeekDay d= WeekDay.MONDAY;
    final WeekDay e= d.nextDay();
    System.out.println("Next day is "+e);
    System.out.println("Prev day is "+d.prevDay());
}
```


Huiswerk / Opdracht





Toledo leerobjecten

- ▣ Onder cursusdocumenten -> Leerobjecten week 2
- ▣ Neem het door, beluister de filmpjes en maak nota's samenvatting voor in het leerverslag