

4 Testen en Logging

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





Testen

Agenda

1. Deel 1: Testen

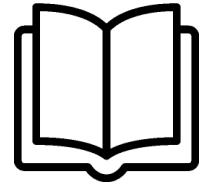
- Het probleem & de oplossing
- JUnit
- De klasse Assert
- Meerdere testklassen combineren



2. Deel 2: Logging

- Logging Frameworks
- Logging Levels
- Logger methoden
- Log Handlers
- Log Formatters

Ondersteunend materiaal



JUnit tutorial:

<https://www.tutorialspoint.com/junit/>

Het probleem



- Belang van testen :
 - Iedere programmeur weet dat hij moet testen
 - Kan manueel, maar efficiënter via testprogramma
 - Slechts weinig programmeurs doen dit ook effectief...
- Waarom niet?
 - te weinig tijd ...(misschien later, als de code stabiel is)
 - te moeilijk ... (de klassen zijn te veel met elkaar gekoppeld)

De oplossing: Unit testen



- Unit testen zijn testprogramma's die meestal **in batch** (geautomatiseerd) uitgevoerd worden.
- Elke test controleert of een bepaalde methode bij een zekere input de verwachte output geeft.
- Je schrijft een **aparte testklasse** voor iedere klasse
 - Voor één methode kan je één ,geen of meerdere tests hebben (afhankelijk van wat er allemaal fout kan gaan)

De oplossing: Unit testen



- **Test Driven Development** (Test First):
schrijf **eerst** de tests, **dan** de code
 - Dan ontbreekt op het einde de tijd niet om te testen
- Test elementaire (kleine, onafhankelijke) functies
 - Dan zijn de tests niet te moeilijk, noch teveel gekoppeld
- **Regression testing**
 - Geautomatiseerde tests kan je zonder inspanning steeds opnieuw draaien. Zo vangen zij ook neveneffecten van nieuwe aanpassingen (bugcorrecties, uitbreidingen) op vroeger geteste functies op (regressies)

Starting from scratch

- Veronderstel deze klasse:

```
public class Rekenmachine {  
    public double sommeer(double a, double b) {  
        return a + b;  
    }  
}
```

Hoe gaan
we dit
testen?



Een eenvoudige testklasse

```
public class TestRekenmachine {  
    public static void main(String[] args) {  
        Rekenmachine calculator = new Rekenmachine();  
        double result = calculator.sommeer(10, 50);  
        if (result != 60) {  
            System.out.println("Foutief resultaat:"  
                                + result);  
        }  
    }  
}
```

Goed
bezig?



Een verbeterde TestRekenmachine

```
public class TestRekenmachine {  
    private int countErrors = 0;  
  
    public void testAdd(){  
        Rekenmachine calculator = new Rekenmachine();  
        double result = calculator.sommeer(10, 50);  
        if (result != 60) {  
            countErrors++;  
            throw new RuntimeException("Foutieve som:"  
                                     + result);  
        }  
    }  
}
```

Al beter, maar
tester moet
namen van
testmethoden
kennen



Het XUnit testing framework

- Oorspronkelijk voor Java, maar ook versies voor andere programmeertalen:
 - C#, VB, Perl, PHP, ADA, Fortran, Delphi, R, Objective-C, JavaScript, Actionscript, ...
- **XUnit** bepaalt de *structuur* van je test cases en voorziet *tools* om de tests uit te voeren.
 - Alternatief test framework: TestNG
(<http://testng.org/doc/>)

- Unit testing framework voor Java
- Wij gebruiken versie 4.12 zie www.junit.org/junit4
→ Javadoc: <http://junit.org/junit4/javadoc/latest/>
- Nog krachtigere versie sinds 10/9/2017:



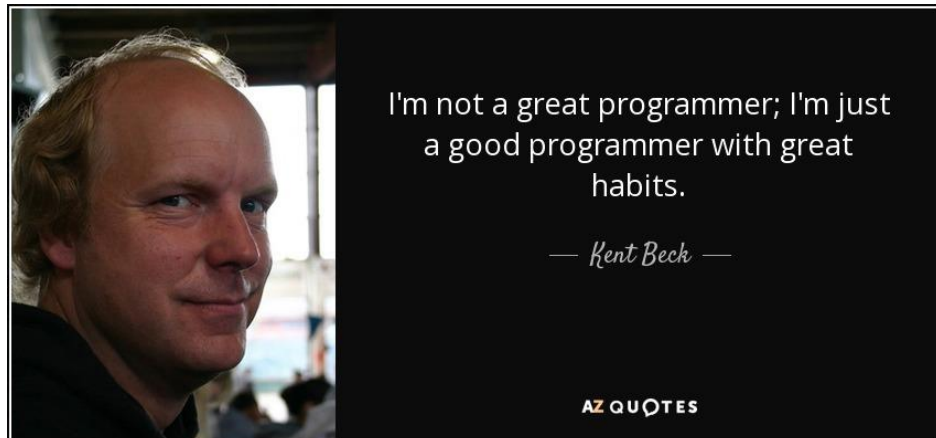
<http://junit.org/junit5/docs/current/user-guide/#overview-what-is-junit-5>

- Eerste versie ontworpen in 1997 door:

- **Erich Gamma**
(the "Gang of Four",
Eclipse IDE)



- **Kent Beck**
(specialist in XP,
TDD, Agile)



Een voorbeeld met JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class TestRekenmachine {
    @Test
    public void sommeer() {
        Rekenmachine rekenmachine = new Rekenmachine();
        double resultaat = rekenmachine.sommee(10, 50);
        assertEquals("De som van 10 en 50 moet 60 zijn"
            , 60, resultaat, 0E-15);    }
}
```



De JUnit annotations

- Wat nodig?

- `import org.junit.*;`
- `junit-4.12.jar` als library toevoegen
- `hamcrest-core-1.3.jar` als library toevoegen



- **@Test** → alle public void methoden met @Test worden uitgevoerd. Je weet niet in welke volgorde
- **@Before** → uitgevoerd voor iedere test
- **@After** → uitgevoerd na iedere test
- **@BeforeClass** → slechts éénmaal voor alle testen in de klasse
- **@AfterClass** → slechts éénmaal na alle testen in de klasse
- **@Ignore** → tijdelijk een test uitsluiten

De JUnit annotations

@Test heeft 2 optionele parameters:

–expected

De volgende test is succesvol:



Weet je ook waarom?

```
@Test(expected = IndexOutOfBoundsException.class)
public void foutje() {
    List<Integer> lijst = new ArrayList<>();
    int eerste = lijst.get(0);
}
```

–timeout

De volgende test faalt:

```
@Test(timeout = 100)
public void oneindig() {
    for(;;);
}
```

TestRekenmachine

```
public class TestRekenmachine {  
    private Rekenmachine rekenmachine;  
  
    @Before  
    public void voorElkeTest() {  
        rekenmachine = new Rekenmachine();  
    }  
  
    @After  
    public void naElkeTest() {  
        rekenmachine = null;  
    }  
  
    @Test  
    public void sommeer() {  
        double resultaat = rekenmachine.sommear(10, 50);  
        assertEquals("De som moet <60> zijn", 60, resultaat, 0E-15);  
    }  
  
    @Test  
    public void vermenigvuldig() {  
        double resultaat = rekenmachine.vermenigvuldig(10, 50);  
        assertEquals("Het product moet <500> zijn", 500,  
            resultaat, 0E-15);  
    }  
}
```

wordt uitgevoerd VOOR elke test

wordt uitgevoerd NA elke test



De klasse Assert

- `junit.framework.Assert`

- Constructor:

- `protected Assert() ;`

Waarom zou men
de constructor
protected maken?



- De assert methoden zijn static en bestaan steeds in 2 overloaded versies:
 - zonder message
 - met message (als eerste parameter, type `String`)

Methoden van de klasse Assert

assertEquals

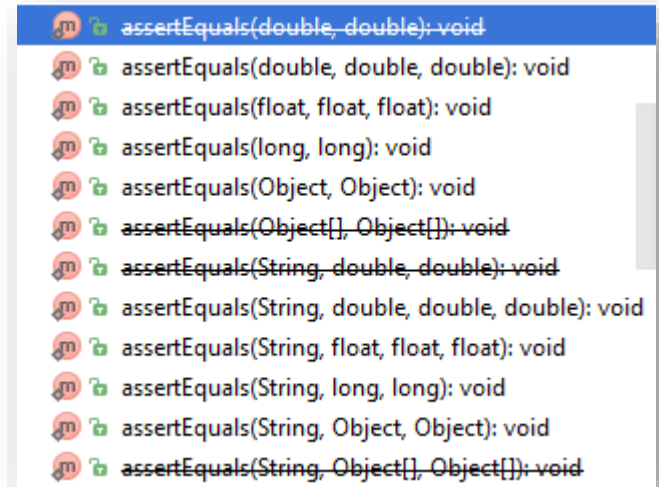
- Method overloading: bestaat voor verschillende de datatypen
- Extra parameter mogelijk om een *foutmelding* mee te geven:

```
assertEquals(int expected, int actual);
```

```
assertEquals(String message, int expected, int actual);
```

- Extra parameter "delta" als *foutenmarge* voor de types `float` en `double`.

```
assertEquals(double expected, double actual,  
double delta);
```



Methoden van de klasse Assert

assertTrue(boolean condition)

assertTrue(String message, boolean condition)

De uitdrukking moet true zijn.

assertFalse(boolean condition)

assertFalse(String message, boolean condition)

De uitdrukking moet false zijn.

assertNotNull(Object object)

assertNotNull(String message, Object object)

Er moet een object zijn.

assertNull(Object object)

assertNull(String message, Object object)

Er mag geen object zijn, m.a.w. de waarde moet null zijn.

Methoden van de klasse Assert

assertSame (Object expected, Object actual)

assertSame (String message, Object expected, Object actual)

De twee parameters moeten naar hetzelfde object verwijzen.

assertNotSame (Object expected, Object actual)

assertNotSame (String message, Object expected,
Object actual)

De twee parameters mogen niet naar hetzelfde object verwijzen.

assertArrayEquals (Object[] expected, Object[] actual)

assertArrayEquals (String message, Object[] expected,
Object[] actual)

De twee arrays bevatten dezelfde elementen.

Er zijn overloaded methoden voor arrays van primitives.

Methoden van de klasse Assert

fail()

fail(String message)

Veroorzaakt het falen van een test.

Elke goede testmethode eindigt met een (of meerdere) **assert...** of **fail** oproepen

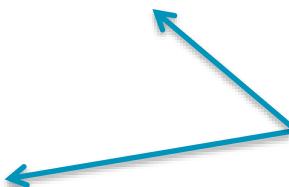


Testcase: Equals en hashCode

- Even herhalen: **equals** en **hashCode** zijn 2 methoden van de klasse **Object**.
- We doen een override van beide methoden als we ZELF de uniciteit van onze klasse willen bepalen.
- **equals**:
 - geeft true als 2 objecten gelijk zijn
 - in **Object**: geeft **true** als 2 objecten hetzelfde fysische adres hebben
- **hashCode**:
 - retourneert een unieke sleutel in de vorm van een **int**
 - in **Object**: retourneert het fysische adres in **int**-vorm

Testcase: Equals en hashCode

```
public class Persoon {  
    private String naam;  
  
    public Persoon(String naam) {  
        this.naam = naam;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Persoon persoon = (Persoon) o;  
        return Objects.equals(naam, persoon.naam);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(naam);  
    }  
}
```



We werken zelf de **equals** en **hashCode** methode van onze klasse **Persoon** uit. We willen het uniek zijn van een **Persoon**-object laten afhangen van het attribuut **naam**

De testklasse (1)

```
public class PersoonTest {  
    private static Persoon pers1;  
    private static Persoon pers2;  
  
    @BeforeClass  
    public static void init() {  
        pers1 = new Persoon("liesa");  
        pers2 = new Persoon("elisa");  
    }  
}
```

De testklasse (2)

@Test

```
public void testEquals() {  
    assertTrue("De namen moeten gelijk zijn",  
        pers1.equals(pers1));  
    assertTrue("De namen moeten gelijk zijn",  
        pers1.equals(new Persoon(string)));  
    assertFalse("De namen moeten verschillen",  
        pers1.equals(pers2));  
    assertFalse("Vergelijken met null moet false geven",  
        pers1.equals(null));  
    assertFalse("Vergelijken met een ander type moet false geven",  
        pers1.equals(new Integer(1)));  
}
```

De testklasse (3)

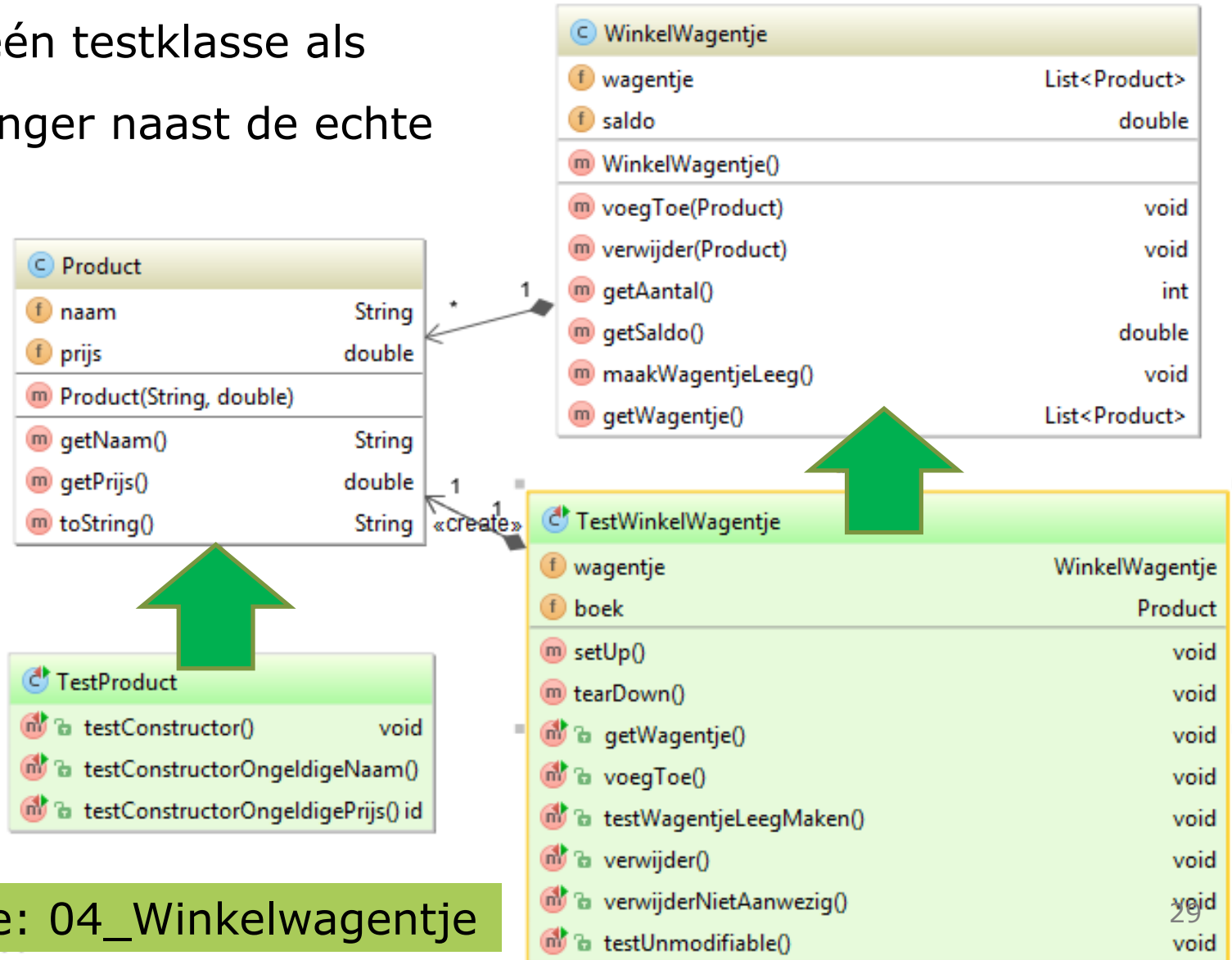
@Test

```
public void testHashCode() {  
    int hashCode1 = pers1.hashCode();  
    Persoon persoon = new Persoon("liesa");  
    int hashCode2 = persoon.hashCode();  
    int hashCode3 = pers2.hashCode();  
  
    assertTrue("De hashcodes moeten gelijk zijn",  
                hashCode1 == hashCode2);  
    assertFalse("De hashcodes moeten verschillend zijn",  
                hashCode1 == hashCode3);  
}
```



Meerdere testklassen combineren

Meestal één testklasse als dubbelganger naast de echte klasse:

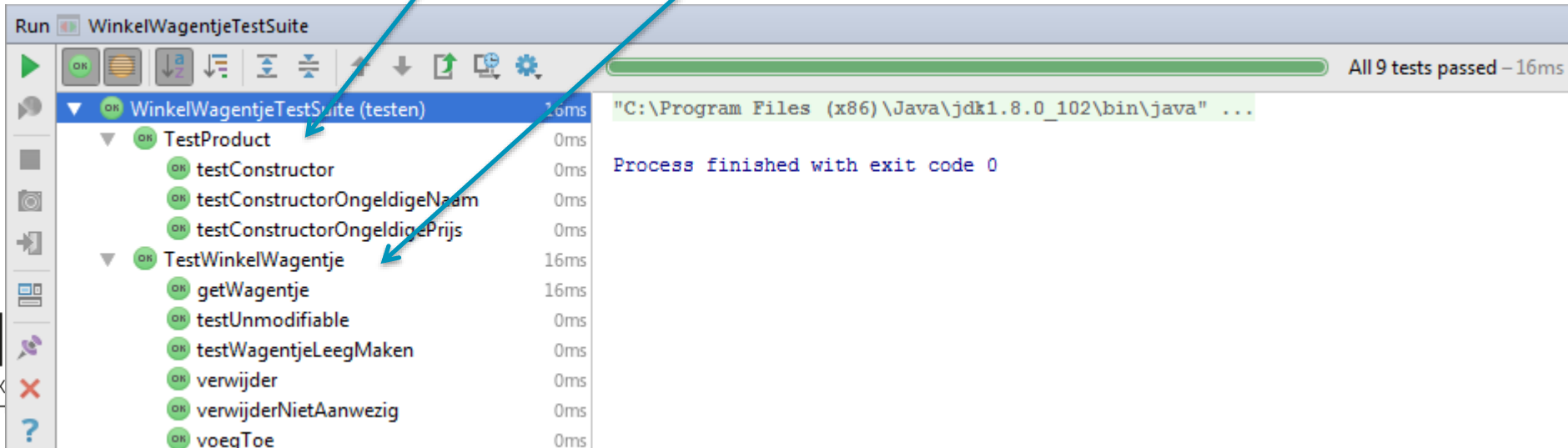


Meerdere testklassen combineren

- Je kan verschillende testklassen samenvoegen in een **TestSuite** en dan runnen in IntelliJ:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestProduct.class , TestWinkelWagentje.class
})
public class WinkelWagentjeTestSuite {
}
```

Alternatief voor TestSuite:
Rechtermuisklik vanuit
Project venster: "Run all
tests"



Meerdere testklassen combineren

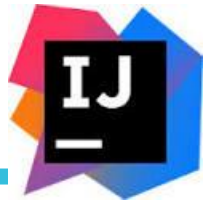
- Je kan ook vanuit een andere klasse de testklassen oproepen:

```
public class WinkelWagentjeTestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(
            WinkelWagentjeTestSuite.class);

        System.out.println("Failures: "
            + result.getFailureCount());
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println("Successful: "
            + result.wasSuccessful());
        System.out.println("Aantal testcases: "
            + result.getRunCount());
        System.out.println("Tijd: " + result.getRuntime()
            + " millisec");
    }
}
```

```
/* OUTPUT: */
Failures: 0
Successful: true
Aantal testcases: 9
Tijd: 15 millisec
```

Result-object
bevat interessante
methoden met
informatie over de
testresultaten.



Geïntegreerd in IntelliJ

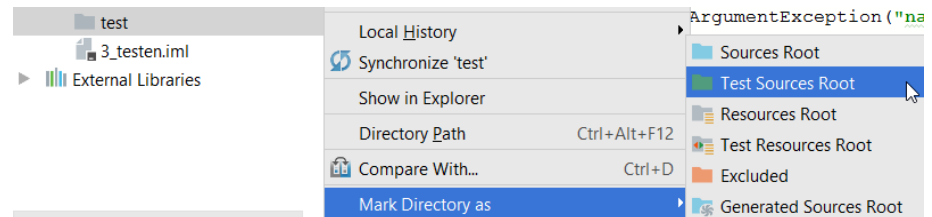
- Bekijk volgende 2-delige demo voor JUnit best practices en ondersteuning in IntelliJ
 - [JUnit 4 with IntelliJ: A quick introduction](#)
 - [JUnit 4 with IntelliJ: Exceptions, Ignore, ...](#)
- We tonen enkele aandachtspunten op de volgende pagina's
- Let op de "test first" benadering die toegepast wordt



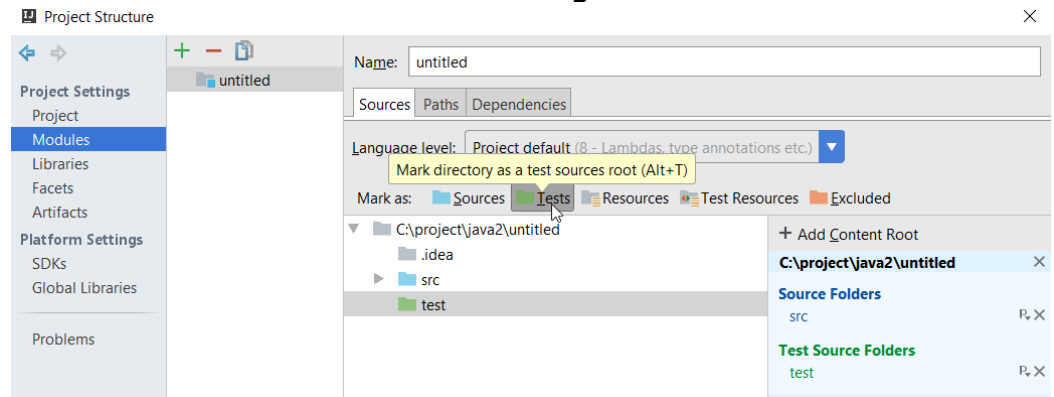
Geïntegreerd in IntelliJ



- Je kan test code in een afzonderlijke map zetten
 - Je testcode blijft dan gescheiden van je productiecode
 - Maak een **test** map
 - Rechtermuismenu op map > *mark directory as* > *Test sources*



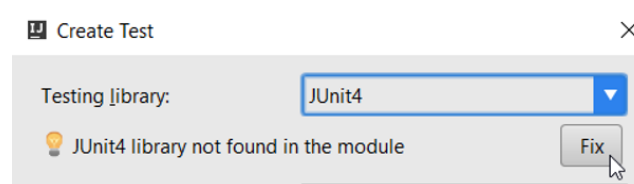
- Ook zichtbaar/configureerbaar onder *File > Project Settings > Modules*





Geïntegreerd in IntelliJ

- In de test map kan je de test klassen in dezelfde packages stoppen als de te testen klassen in src
 - Test klassen binnen dezelfde package hebben toegang tot package private en protected methoden en variabelen van de te testen klassen
- IntelliJ kan testklassen genereren met lege test methoden
 - In te testen klasse: *Navigate>Test*
of ALT+ENTER of CTRL+SHIFT+T
 - Indien nodig stelt IntelliJ voor de JUnit libraries te installeren [Fix]

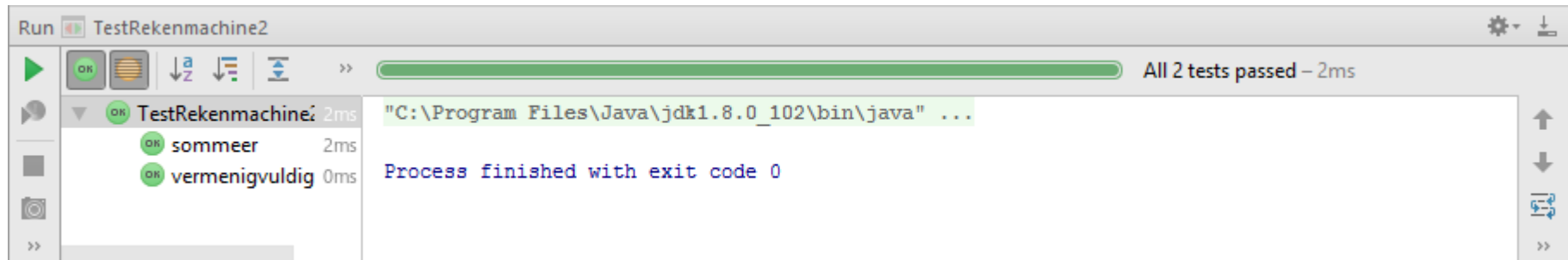


- IntelliJ kan berekenen hoeveel % van je code er via JUnit getest is. Hier bestaan ook specifieke tools voor zoals **JaCoCo** (Java Code Coverage)



Geïntegreerd in IntelliJ

Als je de klasse `TestRekenmachine2` in IntelliJ runt, dan krijg je dit:



Als je in de methode `sommeer` de `+` door een `*` vervangt, dan krijg je dit:

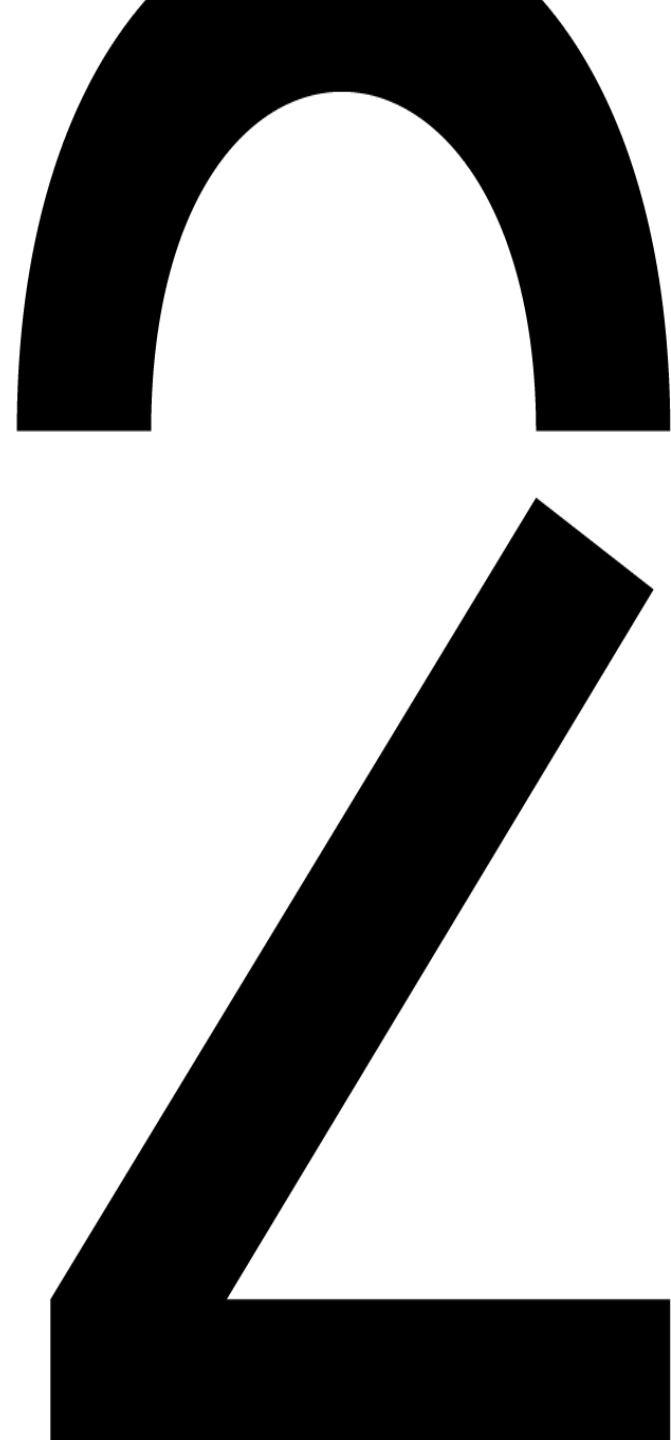


Opdrachten



- Voorbeelden
 - Zie het project "*Voorbeelden_04_Testen_Logging.zip*" bij lesmateriaal, vooral "*04_Winkelwagentje*"
- Groeiproject
 - module 4
(deel 1 en 2: "Testen met JUnit")
- Opdrachten op BB
 - Rekenen
 - Bank
 - Punt





Logging

Agenda

1. Deel 1: Testen

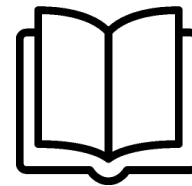
- Het probleem & de oplossing
- JUnit
- De klasse Assert
- Meerdere testklassen combineren



2. Deel 2: Logging

- Logging Frameworks
- Logging Levels
- Logger methoden
- Log Handlers
- Log Formatters

Ondersteunend materiaal



<http://tutorials.jenkov.com/java-logging/index.html>

The screenshot shows the Jenkov.com website with a dark header containing 'Tutorials', 'About', 'RSS', and the 'JENKOV.COM Tech and Media Labs' logo. A cookie notice is present. Below the header is a banner for 'Global CLINICAL SUPPLY Partner. Flexible Solutions. Reliably Supplied.' with details about clinical packaging services, cold chain handling, and cGMP facilities. The main content area is titled 'Java Logging' and includes a sidebar with a table of contents. The main text area features a profile for Jakob Jenkov, a tutorial description, and a note about the scope of the tutorial. The footer contains navigation links: 'All Trails', 'Trail TOC', 'Page TOC', 'Previous', and 'Next'.

Tutorials About RSS

JENKOV.COM
Tech and Media Labs

This site uses cookies to improve the user experience. OK

Global **CLINICAL SUPPLY**
Partner. Flexible Solutions.
Reliably Supplied.

Comprehensive clinical packaging services
Cold chain & specialty handling
8 cGMP facilities & 50+ depots worldwide

Catalent
CLINICAL SUPPLY
LEARN MORE TODAY

Java Logging

1. **Java Logging**
2. Java Logging: Overview
3. Java Logging: Basic Usage
4. Java Logging: Logger
5. Java Logging: Logger Hierarchy
6. Java Logging: Log Levels
7. Java Logging: Formatters
8. Java Logging: Filters

Java Logging

 Jakob Jenkov
Last update: 2014-06-23



This tutorial is about Javas built-in logging API in the `java.util.logging` package. The tutorial explains how to add logging to your Java applications using this API, how to configure it etc. It does not cover *what* you should log, though. What information to log is up to you, depending on what information you need.

The version of the Java logging API covered is the version found in Java 6.

Note: This tutorial does not cover the other popular Java logging API's (Log4J, SLF4J, Apache Commons Logging, LogBack etc.). It only covers Java's built-in logging API, `java.util.logging`.

All Trails Trail TOC Page TOC Previous Next

Logging

- Wegschrijven wat je programma doet.

1. OutputStream: ad hoc logging

```
System.err.println("Openen Stream gefaald");
```

2. Logging framework: uniforme en systematische logging voor je hele toepassing

- Wanneer je logt
- Waarheen je logt
- In welk formaat je logt
- ...



Logging Frameworks

- `java.util.logging`
 - Is onderdeel van JDK
 - Soms afgekort tot *jul*
- Apache log4j 2
- Slf4j / logback



Gebruikt in deze cursus



Meest gebruikt, maar
API staat los van Java

Java.util.logging

1) Maak een **Logger** object aan

- In package `java.util.logging`
- Conventie: 1 `Logger` object per klasse, met de naam van de klasse (kan static gedefinieerd worden)

2) Schrijf boodschappen met behulp van de methoden van het **Logger** object

Voorbeeld: Messenger



Democode:
05_Voorbeelden_logging
> Messenger.java

```
package be.kdg.jul;  
import java.util.logging.Logger;  
import java.util.logging.Level;
```

```
public class Messenger {
```

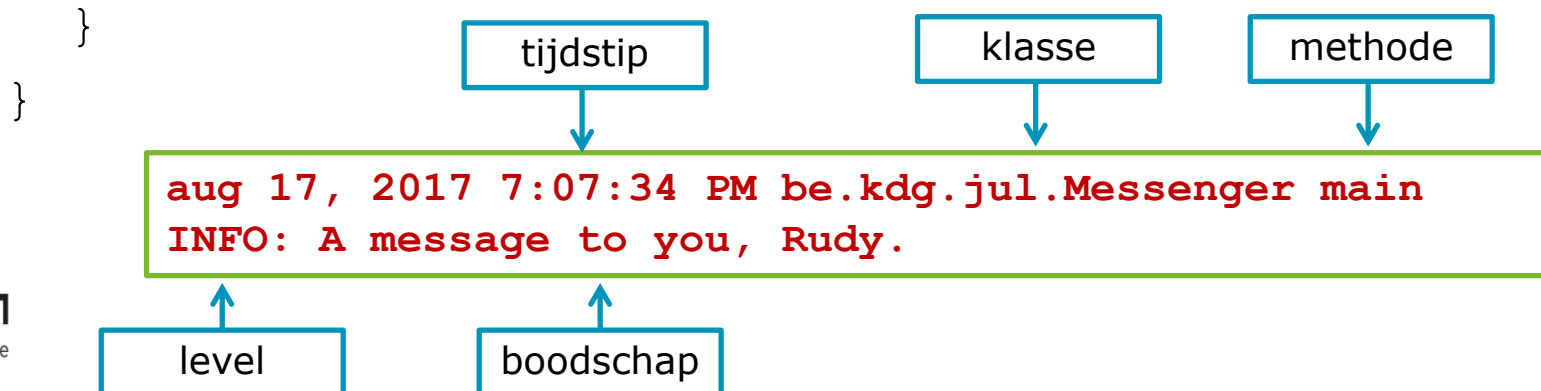
← Naam van de klasse

1)

```
private static final Logger logger =  
    Logger.getLogger("be.kdg.jul.Messenger");
```

2)

```
public static void main(String[] args) {  
    logger.log(Level.INFO, "A message to you, Rudy.");
```



java.util.logging.Level

- Het level duidt aan hoe ernstig een boodschap is.
- Van hoog naar laag:
 - SEVERE
 - WARNING
 - INFO
 - CONFIG
 - FINE
 - FINER
 - FINEST
- Voor elk level is er een shortcut methode. Voorbeeld:
`logger.info("A message to you, Rudy.");`
- Andere logging frameworks gebruiken andere levels

Logging configuratie bestand

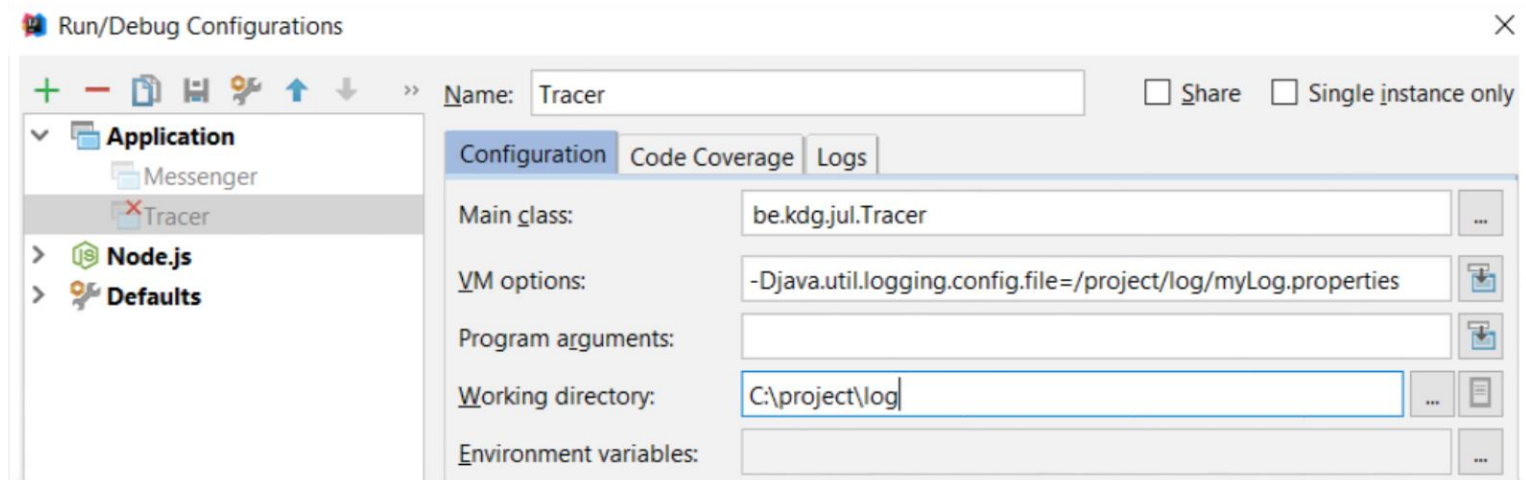
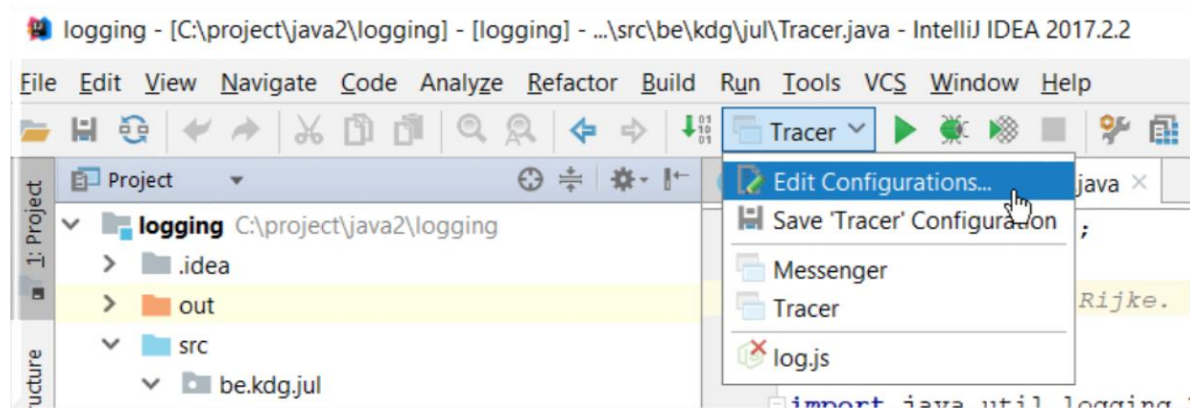
- Standaard gebruikt java logging.properties in de JRE installatie map om logging te configureren:
`$JAVA_HOME/lib/logging.properties`
- Je kan ook zelf een configuratie bestand meegeven met de optie `-Djava.util.logging.config.file`
 - java
`-Djava.util.logging.config.file=/project/log/myLog.properties`
MainClass

Logging configuratie bestand



Democode:
05_Voorbeelden_logging
> Tracer.java
> logging.properties

- In IntelliJ IDEA voeg de optie toe aan de run configuration



Opdracht: configureer logging level



1. Kopieer `$JAVA_HOME/lib/logging.properties` naar je project map
2. Verander in de kopie de regel (regel 29)
`.level= INFO`
naar
`.level= WARNING`
3. Schrap alles na deze regel en bewaar het bestand
4. Refereer naar het logging bestand in de runtime configuratie voor.java
5. Run Messenger opnieuw. Als alles goed is wordt nu niks meer gelogd

Configureer logging level

- Via `logging.properties` bepaalt de uitvoerder hoeveel logging hij wil zien. De ontwikkelde code hoeft hiervoor niet aangepast.
- In `logging.properties` kan je alle logging levels gebruiken, plus
 - ALL: log alle levels
 - OFF: log niks

Configureer package logging level

- Wanneer je de logger naamconventies respecteert, kan je per package (en per klasse) configureren hoeveel gelogd wordt:
 - `.level`: geldt voor alle packages en klassen
 - `be.kdg.level`: geldt voor alle klassen van package `be.kdg`
 - `be.kdg.jul.Messenger.level`: geldt enkel voor logging van deze klasse
- De meest precieze level definitie is deze die toegepast zal worden
- Zo kan je logging selectief verhogen voor packages waarvoor je een probleem vermoedt.

Log Level filteren



Democode:
05_Voorbeelden_logging
> Levels.java
> pkg_filter.properties

```
public class EduMessenger {  
    private static Logger logger =  
        Logger.getLogger("edu.example.EduMessenger");  
  
    public void log(String msg) {  
        logger.log(Level.INFO, msg);  
    }  
}
```

Package: **be.kdg.jul**

Beide loggers staan
standaard ingesteld
op Level.INFO

```
import edu.example.EduMessenger;  
  
public class Levels {  
    private static Logger logger =  
        Logger.getLogger("be.kdg.jul.Levels");  
  
    public static void main(String[] args) {  
  
        logger.log(Level.INFO, "Test be.kdg Level");  
        EduMessenger eduMsg = new EduMessenger();  
        eduMsg.log("Test edu.example Level");  
    }  
}
```

Package: **edu.example**

Log Level filteren

```
handlers= java.util.logging.ConsoleHandler  
.level=WARNING  
edu.level= INFO
```

Pkg filter.properties

Algemeen level = WARNING



C:\dev\java\jdk\8\bin\java ...

aug 21, 2017 5:31:45 PM edu.example.Debugger log
INFO: Test edu.example Level

Alleen voor het edu
package wordt de
INFO boodschap
gelogd

Democode:
05_Voorbeelden_logging
> Levels.java
> pkg_filter.properties

Methode .log



Democode:
05_Voorbeelden_logging
> LoggerMethoden.java

```
String user = "Carolus Magnus";  
String company = "KDG";  
logger.log(Level.INFO, "1. log info boodschap");  
logger.info("2. Verkorte vorm info boodschap");
```

Lange en korte vorm
van loggen

```
logger.log(Level.INFO, "3. Boodschap en exception", new  
    RuntimeException("Caught Exception"));
```

Combinatie met een
Exception

Methode .log



Democode:
05_Voorbeelden_logging
> LoggerMethoden.java

```
String user = "Carolus Magnus";  
String company = "KDG";
```

Gebruik van
variabelen in de
boodschap

```
logger.log(Level.INFO, "4. Concatenatie uitgevoerd voor "  
    + user + " van " + company);  
logger.log(Level.INFO, String.format("5. String.format " +  
    "uitgevoerd voor %s van %s", user, company));
```

Gebruik van **parameters**
en **array** in de boodschap

```
logger.log(Level.INFO, "6. String substitute enkel als " +  
    "INFO actief voor {0} ", user);  
logger.log(Level.INFO, "7. Substitute meerdere parameters "+  
    "met array {0} van {1}", new Object[]{user, company});
```

Log Handlers



Democode:
05_Voorbeelden_logging
> HandlerLevels.java
> fileLogging.properties

- De eerste regel in ons `logging.properties` bestand zegt dat we naar de console loggen:
`handlers= java.util.logging.ConsoleHandler`
- Je kan naar andere plaatsen loggen met andere Handler klassen. Je kan zelf Handlers schrijven of deze uit `java.util.logging` gebruiken: *StreamHandler*, *FileHandler*, *SocketHandler* (naar netwerk server)...
- In Javadoc vind je de configuratie parameters voor elke Handler, die je in `logging.properties` kan gebruiken

FileHandler properties voorbeeld

log naar console EN bestanden

```
handlers= java.util.logging.ConsoleHandler,  
java.util.logging.FileHandler
```

*# log in de home map (%h), met bestandsnamen
java0.log, java1.log ...*

```
java.util.logging.FileHandler.pattern = %h/java%g.log
```

*# log naar een volgend bestand wanneer het log
bestand 1 MB groot is*

```
java.util.logging.FileHandler.limit = 1000000
```

*# Hoe maximum 4 bestanden bij, overschrijf daarna het
eerste bestand*

```
java.util.logging.FileHandler.count = 4
```



Let op met \ in een windows pad (=escape teken in Java).
Gebruik / of \\



Democode:
> 05_Voorbeelden_logging
> HandlerLevels.java
> fileLogging.properties

FileHandler voorbeeld (zonder properties)



Democode:
05_Voorbeelden_logging
> FileDemo.java

Volledige naam
van de klasse



append mode



```
package be.kdg.jul;

import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Logger;

public class FileDemo {
    public static void main(String[] args) throws IOException {
        Logger logger = Logger.getLogger(FileDemo.class.getName());

        FileHandler fileHandler = new FileHandler("mylogging.log", true);
        logger.addHandler(fileHandler);
        logger.setLevel(Level.WARNING);
        logger.info("Information");
        logger.warning("Warning");
    }
}
```

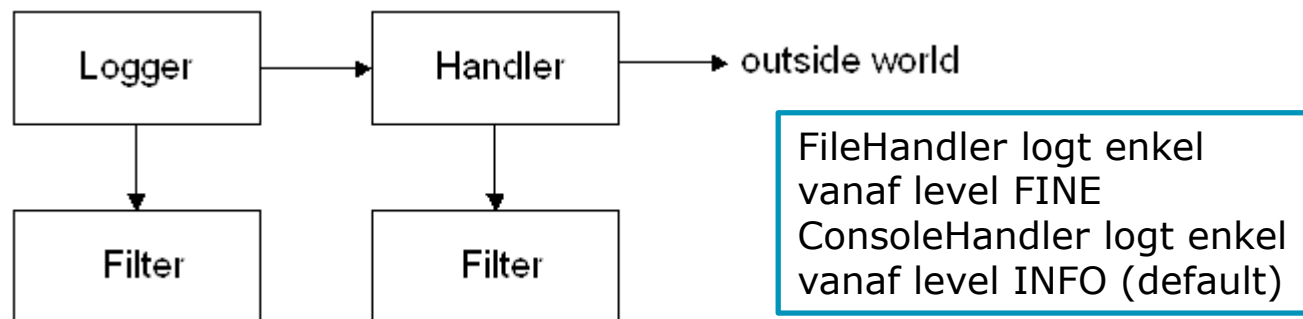
Alles vanaf WARNING en hoger wordt getoond op de console en in XML vorm in het bestand **mylogging.log** opgeslagen. Het bestand bevindt zich op het project directory niveau.

Log Handlers

Voor elke handler kan je een Level specificeren
Boodschappen lager dan dit level worden door de handler
genegeerd

```
handlers= java.util.logging.ConsoleHandler,  
java.util.logging.FileHandler
```

```
java.util.logging.FileHandler.level= FINE
```



Je kan ook je eigen specifieke Filters (java.util.logging.Filter interface) schrijven en toevoegen

Log Handlers

```
public static void main(String[] args) {  
    logger.log(Level.FINER, "A message to you, Rudy.");  
    logger.log(Level.FINE, "Message in a bottle.");  
    logger.log(Level.INFO, "Kleine boodschap.");  
}
```

Java0.log



C:\dev\java\jdk\8\bin\java ...

aug 22, 2017 10:16:58 AM be.kdg.jul.Filter main
INFO: Kleine boodschap.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
<!DOCTYPE log SYSTEM "logger.dtd">  
<log>  
  <record>  
    <date>2017-08-22T10:16:58</date>  
    <millis>1503389818957</millis>  
    <sequence>1</sequence>  
    <logger>be.kdg.jul.Messenger</logger>  
    <level>FINE</level>  
    <class>be.kdg.jul.Filter</class>  
    <method>main</method>  
    <thread>1</thread>  
    <message>Message in a bottle.</message>  
  </record>  
  <record>  
    <date>2017-08-22T10:16:58</date>  
    <millis>1503389818973</millis>  
    <sequence>2</sequence>  
    <logger>be.kdg.jul.Messenger</logger>  
    <level>INFO</level>  
    <class>be.kdg.jul.Filter</class>  
    <method>main</method>  
    <thread>1</thread>  
    <message>Kleine boodschap.</message>  
  </record>  
</log>
```

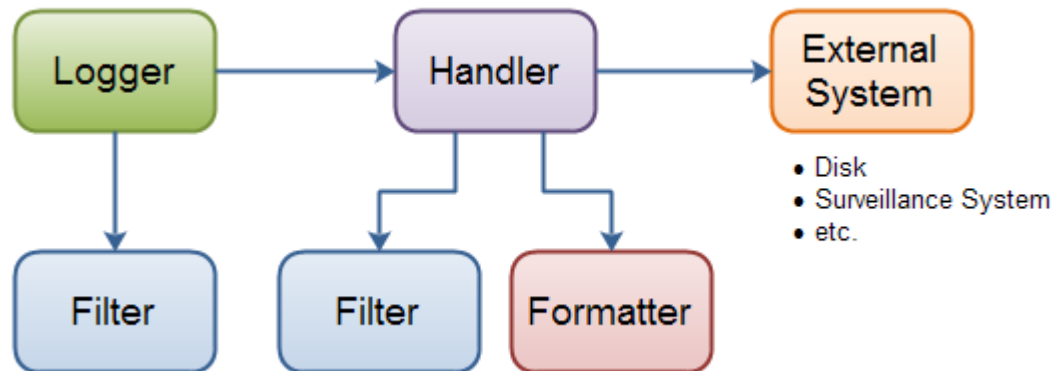
Democode:

05_Voorbeelden_logging
> HandlerLevels.java
> fileLogging.properties



Log Formatters

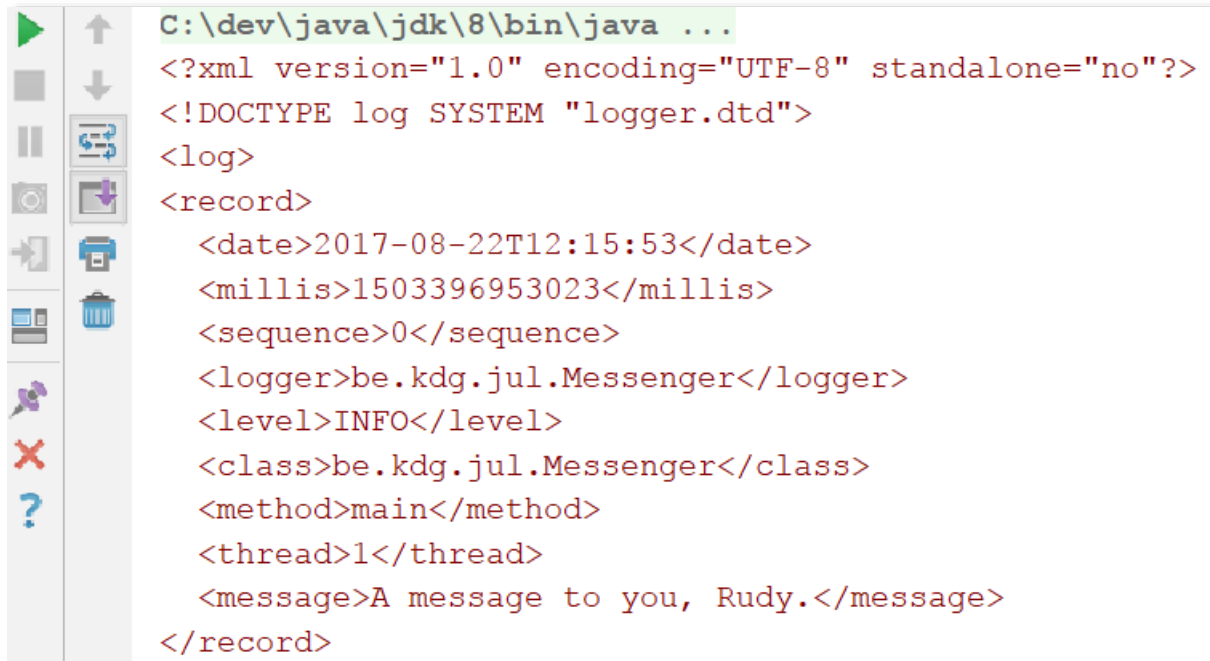
- Merk het verschil in formaat op tussen beide logs op vorige slide
 - **ConsoleHandler** gebruikt default SimpleFormatter
 - **FileHandler** gebruikt default XMLFormatter



- Je kan zelf een **Formatter** schrijven. Extend hiervoor de abstracte klasse `java.util.logging.Formatter`. Voeg dan je **Formatter** toe aan een **Handler** in het `logging.properties` bestand.

Log Formatters

```
handlers= java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.formatter=
    java.util.logging.XMLFormatter
```

A screenshot of a Java IDE window. The title bar shows the path 'C:\dev\java\jdk\8\bin\java ...'. The main text area displays an XML log entry. The XML is well-formatted with red text on a white background. It starts with an XML declaration, followed by a DOCTYPE declaration for 'log SYSTEM "logger.dtd"'. The root element is '<log>'. Inside, there is a '<record>' element containing several sub-elements: '<date>2017-08-22T12:15:53</date>', '<millis>1503396953023</millis>', '<sequence>0</sequence>', '<logger>be.kdg.jul.Messenger</logger>', '<level>INFO</level>', '<class>be.kdg.jul.Messenger</class>', '<method>main</method>', '<thread>1</thread>', and '<message>A message to you, Rudy.</message>'. The record element is closed with '</record>'. On the left side of the IDE, there is a vertical toolbar with various icons for running, debugging, and other IDE functions.

```
C:\dev\java\jdk\8\bin\java ...
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
  <record>
    <date>2017-08-22T12:15:53</date>
    <millis>1503396953023</millis>
    <sequence>0</sequence>
    <logger>be.kdg.jul.Messenger</logger>
    <level>INFO</level>
    <class>be.kdg.jul.Messenger</class>
    <method>main</method>
    <thread>1</thread>
    <message>A message to you, Rudy.</message>
  </record>
```

Opdrachten



- Groeiproject
 - module 4
(deel 3: "Logging")
- Opdrachten op BB
 - Logging
 - Computer
- Zelftest!

