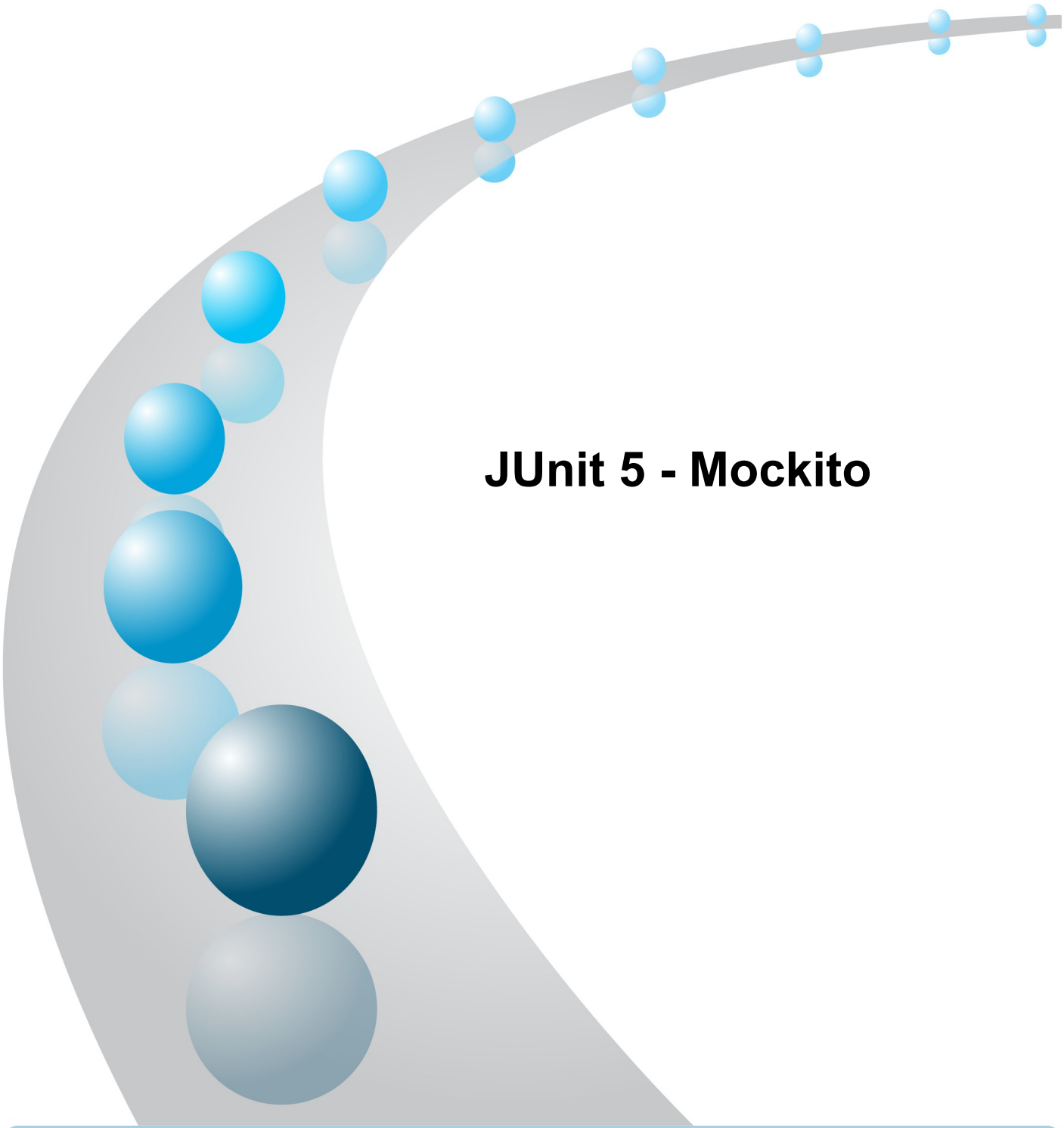
Noël Vaes

Java Trainer & Consultant



JUnit 5 - Mockito

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Correspondentie inzake overnemen of reproductie kunt u richten aan:

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

04/04/2019

Inhoudsopgave

Hoofdstuk 1: JUnit

1.1 Inleiding

Het grondig testen van software is een belangrijk onderdeel bij de ontwikkeling ervan. Als programmeur hebben we vaak de neiging deze activiteit achterwege te laten vanwege tijdsgebrek of omdat het schrijven van nieuwe functionaliteit ons gewoon meer aantrekt dan het testen van de reeds geschreven code. Het testen laten we dan over aan de mensen van de testafdeling, of in het ergste geval: de klant!
Het consequent testen van de verschillende modules lijkt op het eerste zicht tijdrovend maar deze investering verdient zich op langere termijn terug. De software is namelijk veel stabieler en bevat veel minder onverwachte nevenwerkingen. De tijd die men achteraf steekt in het zoeken naar diep verborgen *bugs* is daardoor veel korter.

Bij het testen van software onderscheiden we drie vormen:

1. ***Unit test***: hierbij worden de afzonderlijke modules of software-eenheden op zich getest.
2. ***Functional test***: hierbij wordt een stuk functionaliteit getest. Dit impliceert doorgaans de samenwerking tussen verschillende modules.
3. ***Integration test***: hierbij wordt het gehele systeem getest van het begin tot het einde.

In objectgeoriënteerde talen is een module of eenheid het object, of de klasse waar het object een instantie van is. Dit impliceert dus dat we eigenlijk elke klasse die we maken afzonderlijk moeten testen. Bij *Extreme Programming* gaat men zelfs nog een stap verder en begint men eerst met het schrijven van de test om daarna een klasse te maken die aan de testvoorwaarden voldoet. Het hele ontwikkelingsproces wordt hier voortgestuwd door de testen (*test driven development*).

Dat klinkt allemaal mooi in theorie, maar om programmeurs aan te zetten tot het effectief schrijven van de nodige tests, is er een werkwijze nodig waarbij het maken van deze tests eenvoudig en snel is.

Om aan die verzuchting tegemoet te komen bestaan er *test frameworks* die een aantal taken op zich nemen. In de Java-wereld is het meest gekende en meest gebruikte het *open source framework* ***JUnit***. Dit is te vinden op volgende website: ***www.junit.org.*** *JUnit* is in eerste instantie een *framework* voor het testen van **J**ava **Unit**s. De focus ligt dus op *unit testing*.

In deze cursus nemen we dit *framework* onder de loep. We gebruiken hiervoor versie 5 van *JUnit*. Deze versie draagt ook de naam *Jupiter*.

1.2 Mijn eerste test

Tijd om zelf onze eerste test te schrijven.

Bij *unit testing* is het de bedoeling dat men iedere *unit* afzonderlijk kan testen. Zo'n *unit* is in dit geval een klasse. We maken daarom eerst een eenvoudige klasse waarvoor we nadien een test gaan schrijven. Ja, hier komt hij weer: de "***Hello World***":

```
package eu.noelvaes.junit;

public class HelloWorld {
   public String sayHello() {
      return "Hello World";
   }
```

```
}
```

Deze klasse heeft een methode `sayHello()` die de string ***"Hello World"*** teruggeeft.

Voor deze klasse gaan we nu een testklasse schrijven. Het is gebruikelijk deze testklasse onder te brengen in hetzelfde pakket. Dat maakt dat de testklasse toegang krijgt tot alle *members* met *package* toegangsniveau. Doorgaans zet men de broncode van de testklassen wel in een andere broncodemap (*test*).

Een testklasse is een gewone klasse die voorzien is van een aantal testmethoden. Deze testmethoden krijgen de annotatie `@Test:`

```
package eu.noelvaes.junit;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;

public class HelloWorldTest {
   @Test
   public void testSayHello() {
      HelloWorld hello = new HelloWorld();
      String answer = hello.sayHello();
      Assertions.assertEquals("Hello World", answer);
   }
}
```

In de testmethode maken we eerst een instantie van de klasse `HelloWorld`. Vervolgens roepen we de methode `sayHello()` op en bewaren het resultaat in een variabele. Ten slotte testen we met de methode `assertEquals()` of het resultaat overeenkomt met het verwachte resultaat.

We voegen verder aan ons project nog een modulebeschrijving toe:

module-info.java

```
module eu.noelvaes.junit {
}
```

Om deze test nu uit te voeren moeten we gebruikmaken van de *testrunner* van *JUnit.* Dit kan het makkelijkst via de geïntegreerde *plugin* in de IDE, via ANT of *Maven*.

1.3 Integratie in de ontwikkelomgeving

JUnit kan afgehaald worden op de site *www.junit.org.* Doorgaans is dit niet nodig daar *JUnit* geïntegreerd is in de meeste gangbare IDE's zoals *Eclipse, NetBeans, IntelliJ* enzovoort. We kunnen dus gewoon gebruikmaken van deze ingebouwde mogelijkheid. Bovendien bevatten deze IDE's speciale *JUnit plugins* die de resultaten van de testen grafisch zichtbaar maken. In deze paragraaf zullen we de integratie in *Maven* meer in detail bekijken. Voor het verdere verloop van de cursus kies je één van deze twee. *Maven* geniet evenwel de voorkeur.

1.3.1 Integratie met *Maven*

JUnit is standaard geïntegreerd in *Maven*. Voor de uitvoering van de testen wordt er beroep gedaan op de *SureFire plugin*. Deze zal automatisch alle testen uitvoeren die gevonden worden in de map ***src/test/java.***

Voor *JUnit 5* in combinatie met JDK 11 moeten we evenwel gebruikmaken van een recente

versie van de *SureFire plugin* (3.0.0 of hoger).

Verder dienen we een *dependency* voor *JUnit* 5 aan het project toe te voegen. Dit ziet er in de POM dan als volgt uit:

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>3.0.0-M3</version>
      </plugin>
   </plugins>
</build>

<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.2.0</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Vermits we *JUnit* enkel tijdens het testen nodig hebben, zetten we de *scope* op ***test***.

Het compileren en het uitvoeren van de testen maakt deel uit van de *default lifecycle*. In de volgende tabel geven we de verschillende *goals* in deze *lifecycle* weer:

Fase	*Omschrijving*
`validate`	Nagaan of het project geldig is en alle noodzakelijke informatie beschikbaar is.
`generate-sources`	Genereer automatisch broncode.
`process-sources`	Bewerk de gegenereerde broncode.
`generate-resources`	Genereer automatisch andere *resources.*
`process-resources`	Bewerk de gegenereerde *resources* en kopieer ze naar de doelmap.
`compile`	Compileer de broncode.
`process-classes`	Bewerk eventueel de *bytecode*.
`generate-test-sources`	**Genereer automatisch de test-broncode.**
`process-test-sources`	**Bewerk de gegenereerde test-broncode.**
`generate-test-resources`	**Genereer extra *resources* voor de test.**
`process-test-resources`	**Bewerk de gegenereerde test-*resources* en kopieer ze naar de doelmap.**
`test-compile`	**Compileer de test-broncode.**
`test`	**Voer de *unit*-testen *(JUnit* of *TestNG)* uit.**
`prepare-package`	Voorbereiding op het maken van het pakket.
`package`	Maken van het pakket (JAR,WAR, EAR …).
`pre-integration-test`	Voorbereiding op de integratietest.
`integration-test`	Voer de integratietest uit.
`post-integration-test`	Nabewerking van de integratietest; opkuis bijvoorbeeld.
`verify`	Controleer de geldigheid van het pakket.
`install`	Voeg het pakket toe aan de lokale *repository* voor eigen lokaal gebruik.
`deploy`	Voeg het pakket toe aan de globale *repository* voor algemeen gebruik. Dit veronderstelt wel dat men voldoende rechten heeft om dit te doen.

De testfase kan expliciet uitgevoerd worden met het volgende commando:

```
mvn test
```

Afzonderlijke testen kunnen als volgt uitgevoerd worden:

```
mvn test -Dtest=TestName
```

Opdracht 1: Een Maven-project maken

In deze opdracht maken we een *Maven*-project en voegen we hierin een testklasse toe.

- Maak een nieuw *Maven*-project, eventueel met behulp van je IDE.
- Voeg de volgende *plugin* en *dependency* toe:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>eu.noelvaes</groupId>
   <artifactId>JUnit</artifactId>
   <version>5</version>

   <properties>
      <maven.compiler.source>11</maven.compiler.source>
      <maven.compiler.target>11</maven.compiler.target>
      <project.build.sourceEncoding>
         UTF-8
      </project.build.sourceEncoding>
   </properties>

   <build>
      <plugins>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M3</version>
         </plugin>
      </plugins>
   </build>

   <dependencies>
      <dependency>
         <groupId>org.junit.jupiter</groupId>
         <artifactId>junit-jupiter-engine</artifactId>
         <version>5.2.0</version>
         <scope>test</scope>
      </dependency>
   </dependencies>
</project>
```

- Voeg de volgende klasse toe in de map ***src/main/java/eu/noelvaes/junit***:

```
package eu.noelvaes.junit;

public class HelloWorld {
   public String sayHello() {
      return "Hello World";
   }
}
```

- Voeg vervolgens de volgende testklasse toe in de map ***src/test/java/eu/noelvaes/junit***:

```
package eu.noelvaes.junit;
import org.junit.jupiter.api.Test;

public class HelloWorldTest {
   @Test
   public final void testSayHello() {
      HelloWorld hello = new HelloWorld();
      String answer = hello.sayHello();
      Assertions.assertEquals("Hello World", answer);
```

```
    }
}
```

- Voer de test uit met het volgende commando:
  `mvn test`
- Introduceer een fout in de test en voer de test opnieuw uit.

1.4 De levenscyclus van een testklasse

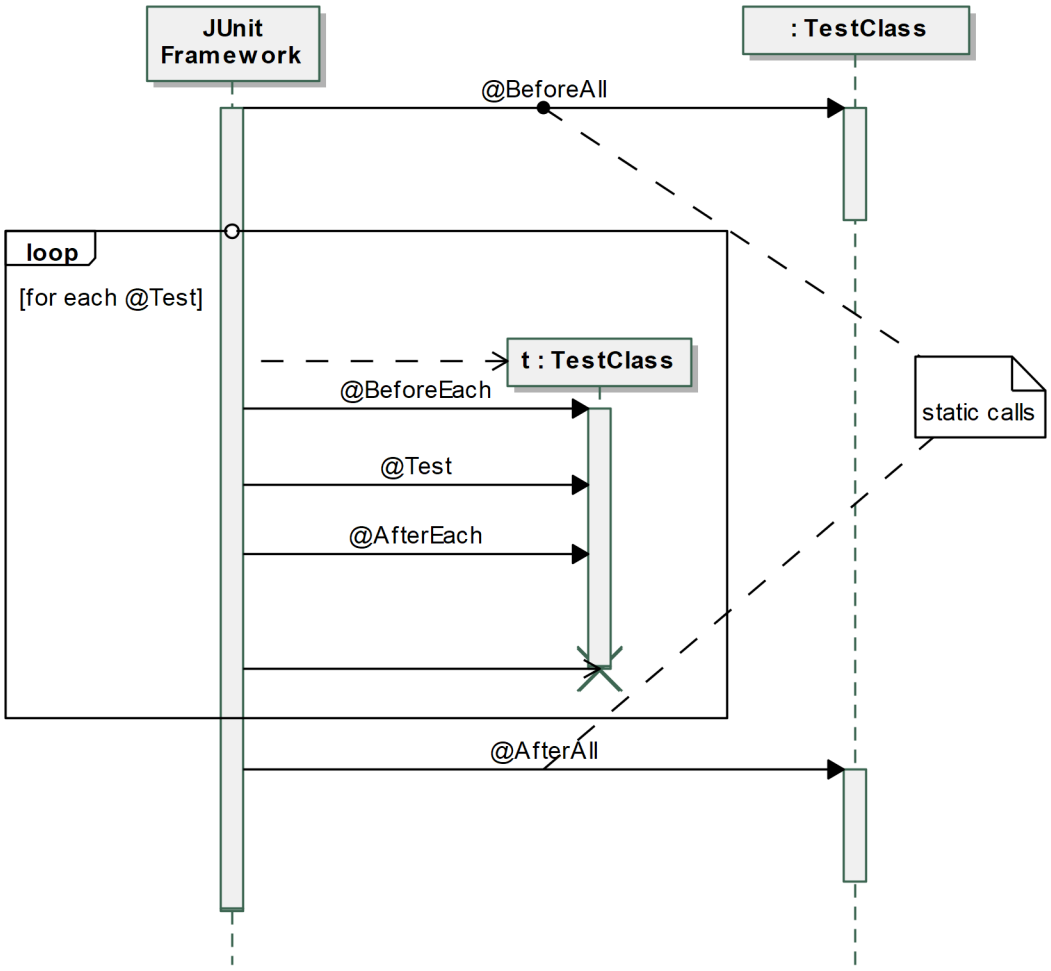
Het uitvoeren van de testmethoden in de testklasse wordt gedaan door het *JUnit framework*, meer bepaald door de *engine* van het *framework*. Deze gaat bij een testklasse op zoek naar methoden die de annotatie `@Test` hebben. Iedere testmethode wordt beschouwd als een afzonderlijke geïsoleerde test. Indien er meerdere testmethoden in dezelfde klasse zitten, mogen die geen invloed van elkaar ondervinden. Om die reden wordt bij *JUnit* voor iedere testmethode telkens een nieuwe instantie van de testklasse gemaakt en vervolgens wordt de testmethode uitgevoerd. De volgorde waarin de verschillende testen worden uitgevoerd, is overigens onbepaald.

Meerdere testen kunnen evenwel nood hebben aan dezelfde initialisatie- en opruimcode. Om die reden kunnen er extra methoden voorzien worden met de annotaties `@BeforeEach` en `@AfterEach`. Deze methoden worden respectievelijk voor en na het uitvoeren van iedere test door *JUnit* opgeroepen.

Indien men toch gemeenschappelijke code nodig heeft die éénmalig voor en na het geheel van alle testen wordt uitgevoerd, kan men gebruikmaken van *static* methoden met de annotaties `@BeforeAll` en `@AfterAll`. Deze methoden worden door *JUnit* voor en na de reeks van testen uitgevoerd.

We merken verder nog op dat de naam van de testmethoden onbelangrijk is. Ze mogen wel niet *private* of *static* zijn en ze mogen geen *return*-waarde hebben. Eventueel kunnen er aan de constructor en de methoden bepaalde argumenten meegegeven worden. We komen hier later nog op terug.

We geven dit alles schematisch weer in het onderstaande sequentiediagram:



De algemene code van een test zou er daarom als volgt kunnen uitzien:

```
package eu.noelvaes.junit;
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.TestInstance.Lifecycle;

public class TestClass {

   public TestClass() {
      System.out.println("constructor");
   }

   @BeforeAll
   public static void beforeAll() {
      System.out.println("beforeAll");
   }

   @BeforeEach
   public void beforeEach() {
      System.out.println("beforeEach");
   }

   @Test
   public void test1() {
```