**1Toets/programmeeropdracht:**

* Creeer een nieuw Java project met Maven (via de command-line of met IntelliJ)
* Importeer het project in je IDE
* Voeg een of meerdere Maven dependencies toe
* Bouw volgens specificatie een of twee classes met enkele methoden waarin input gevalideerd moet worden gebruik makend van Java Exceptions.
* Bouw JUnit testcases voor een selectie van de methoden. De testcases moeten zorgen voor 100% test coverage van de gegeven methoden.
* Voer de main-methode van de applicatie uit via Maven en run de test cases via Maven.
* Verbouw de applicatie zodat deze gebruik maakt van de Thread API door gebruik van de class Thread, interface Runnable en het keyword synchronized.
* Verbouw de applicatie zodat het een client-server applicatie is die gebruik maakt van het RMI protocol: de classes UnicastRemoteObject, Registry en RemoteException en de interfaces Remote en Serializable.

**User story 1: Maven**

**Beschrijving:**

*As a student, I need to learn Maven so I can build, test and run applications with a script outside an IDE to prepare for working in a continuous integration environment.*

**Apache Maven** is een [softwaregereedschap](https://nl.wikipedia.org/wiki/Software) voor Java-projectmanagement en geautomatiseerde softwarebouw. Het is gelijk in functionaliteit aan het gereedschap [Apache Ant](https://nl.wikipedia.org/wiki/Apache_Ant) (en iets minder aan [PHP](https://nl.wikipedia.org/wiki/PHP)'s PEAR en [Perl](https://nl.wikipedia.org/wiki/Perl_(programmeertaal))'s CPAN), maar heeft een simpele bouw configuratie, gebaseerd op de taal [XML](https://nl.wikipedia.org/wiki/Extensible_Markup_Language). Maven wordt gefaciliteerd door de [Apache Software Foundation](https://nl.wikipedia.org/wiki/Apache_Software_Foundation), waar het voorheen een onderdeel was van het Jakarta Project.

Maven gebruikt een "Project Object Model" (POM) om het software ontwikkeltraject te sturen. In de POM staan verder de afhankelijkheden met andere modules en componenten, waaruit de volgorde van bouwen bepaald wordt. In de POM kunnen naast de gebruikelijke stappen als compileren en samenvoegen voor distributie, extra acties gedefinieerd worden die het ontwikkelproces kunnen ondersteunen. Voorbeelden hiervan zijn automatisch testen, (statische) code verificatie en analyse van "Code Coverage" door de testen.

Een belangrijk aspect van Maven is de zogenaamde "repository" waarin verschillende versies van componenten opgeslagen zijn. Dit kunnen componenten zijn waarvan de te bouwen software rechtstreeks afhankelijk is. Ook kunnen dit componenten zijn die het bouwproces zelf ondersteunen. Maven biedt ondersteuning om de repository automatisch te vullen met versies die op het internet aangeboden worden, via Apache en andere organisaties.

<https://maven.apache.org/download.cgi>

<https://maven.apache.org/install.html>

In de powerpoint van school staat ook veel info (Week 1b; Maven).

**GroupID** = projectnaam

**Artifactid** = naam van jaar zonder versienummer.

**Version** = versienummer.

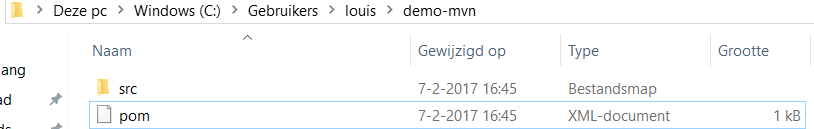
**Acceptatiecriteria:**

* Kan externe libraries toevoegen als Maven dependency.

Uitleg/uitwerking:

Wanneer je een externe library wil toevoegen aan je maven project ga je naar je

Ga je naar je door maven gecreëerde map. In dit geval is dat <demo-mvn>.



Open je Pom.xml bestand in bijvoorbeeld kladblok.



Hierin staat <dependencies> daarin staan alle dependencies waarvan jou project afhankelijk is. We willen aan deze een nieuwe <dependency> toevoegen.

<dependency>

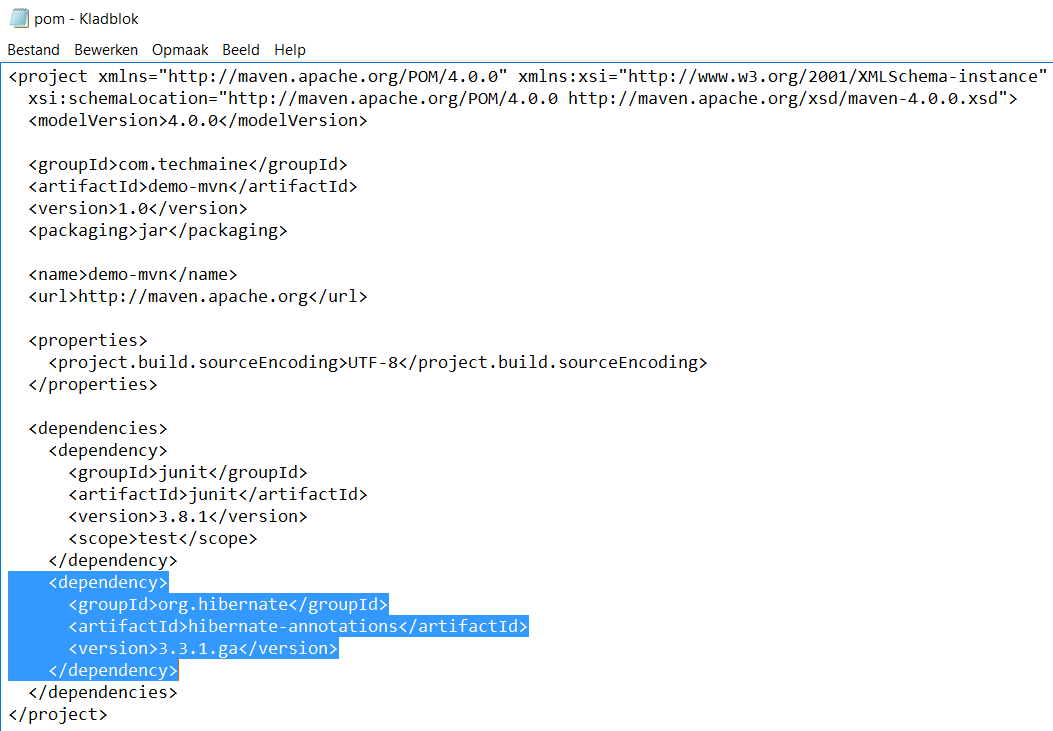
<groupId>org.hibernate</groupId>

<artifactId>hibernate-annotations</artifactId>

<version>3.3.1.ga</version>

</dependency>

Dit voeg je toe aan je POM.xml



Je kan aan je dependencies verschillende <scopes> toevoegen.

**<scope>Compile</scope>**

This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.

**provided**

This is much like compile, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope provided because the web container provides those classes. This scope is only available on the compilation and test classpath, and is not transitive.

**runtime**

This scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.

**test**

This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive.

**system**

This scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

**import** *(only available in Maven 2.0.9 or later)*

This scope is only supported on a dependency of type pom in the <dependencyManagement> section. It indicates the dependency to be replaced with the effective list of dependencies in the specified POM's <dependencyManagement> section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

* Kan versies van dependencies aanpassen.

Uitleg/uitwerking:

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-annotations</artifactId>

<version>3.3.1.ga</version>

</dependency>

In deze dependency is de versie van deze library 3.3.1.ga . Stel deze library word geupdate en er word een nieuwe versie uitgebracht moet je dus in je POM.xml bestand aangeven dat je een nieuwe versie wil gaan gebruiken.

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-annotations</artifactId>

<version>3.3.2.ga</version>

</dependency>

* Kan Maven command-line gebruiken voor het runnen en unit-testen van een applicatie.

Uitleg/uitwerking:

Eerst moet een Maven project aangemaakt worden; doe dit door command line: mvn archetype:generate uit te voeren.

Hierop volgend dien je het aangemaakte project te importeren in IntelliJ/Eclipse. Dit doe je door in het hoofdmenu te kiezen voor Import Project. Vervolgens kies je de map waar het Maven project staat en kies je voor ‘open’.

Klik een aantal keer door en het project wordt geïmporteerd.

Nadat het project geïmporteerd is is het tijd om een aantal namen aan te passen. Hernoem de map binnen map ‘Java’ (waarschijnlijk com.techmaine oid) naar ‘com.oose’.

Plak vervolgens de volgende code binnen de tag <plugin> in het pom.xml bestand:

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.0.2</version>

<configuration>

<source>1.6</source>

<target>1.6</target>

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<version>1.1-beta-1</version>[

<artifactId>exec-maven-plugin</artifactId>

<configuration>

<mainClass>com.oose.App</mainClass>

</configuration>

</plugin>

In de bovenstaande code staat de tag <mainClass>. Hernoem dit naar de mapstructuur + naam van je main klasse.

Ten slotte kan de applicatie uitgevoerd worden door ‘mvn package’ en vervolgens ‘mvn exec:java’ uit te voeren in de command line.

**Unit testing**

Neem de volgende regels code op boven een testclass:

import junit.framework.Assert;

import org.junit.Test;

In de klasse zelf dien je een tag te geven en daaronder een methode die een test uitvoert. Zie het volgende voorbeeld:

@Test

public void testPrintHelloWorld() {

Assert.*assertEquals*(App.*getHelloWorld*(), "Hello World");

}

Onthoudt ook de naam van de testklasse; deze kun je gebruiken in Maven om een specifieke test uit te voeren.

In Maven kun je m.b.v. de command line één of meerdere unit tests uitvoeren. Met de volgende code kun je een specifieke test uitvoeren; mvn -Dtest=classname test .Hier hoort classname de naam van je klasse te zijn.

Wanneer je alle testen in je project wilt uitvoeren dien je de ‘-Dtest=’ weg te laten, waarna je dus mvn test overhoudt.

* Kent de lifecycle van Maven met minimaal de goals *clean*, *compile*, *test* en *package*.

Uitleg/uitwerking:

* validate - validate the project is correct and all necessary information is available
* compile - compile the source code of the project
* test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package - take the compiled code and package it in its distributable format, such as a JAR.
* verify - run any checks on results of integration tests to ensure quality criteria are met
* install - install the package into the local repository, for use as a dependency in other projects locally
* deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

Default lifecycle: validate, compile, package, install. Alleen de laatste hoeft uitgevoerd te worden; Maven voert automatisch de vorige stappen uit de build lifecycle uit.

Clean: Verwijderd eerst alles in de target directory, en begint vervolgens met het opvolgende commando ( install, deploy).

Voor gedetailleerde uitleg en onderdelen van elk lifecycle onderdeel zie: <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

Volgens de opdracht dien je minstens clean, compile, test en package te kennen:

* Clean: leegmaken van de target directory.
* Compile: het compileren van de source code.
* Test: uitvoeren van (unit) tests.
* Package: gecompileerde code verpakken in bijvoorbeeld een .JAR.

* Hoeft geen rekening te houden met multi-modules

**Maven Archetypes**

Maven kent verschillende archetypes. Archetypes zijn als het ware een template voor een project. De archetypes zijn hier te vinden:

<https://maven.apache.org/guides/introduction/introduction-to-archetypes.html>

**User story 2: Exceptions**

**Beschrijving:**

*As a student, I need to learn checked and unchecked exceptions so I can handle runtime errors properly.*

# Unchecked/Checked exceptions

**Unchecked exceptions**

Unchecked exceptions zijn fouten in het programma(bugs). Deze exceptions zijn subclasses van de class RunTimeException.

Methodes hoeven hier niet rekening mee te houden, aangezien het bugs zijn en als je van de bug af weet dan kun je beter de bug fixen ipv methods aanpassen.

**Checked exceptions**

Checked exceptions zijn fouten buiten het programma, zoals invalid user inputs, database problemen, netwerk storing, en absente bestanden.

Checked exceptions zijn subclasses van de class Exception

Methodes moeten hier rekening mee houden.

**Wanneer moet je exceptions gebruiken**

Als je denkt dat er een kans is dat een stukje code errors gaat geven doordat bijvoorbeeld een filepath/schrijf niet meer bestaat of dat er foute user input kan gebeuren. Wanneer er een exception error komt die niet wordt opgevangen in een exception kapt je hele programma ermee met een “Uncaught Exception…”.

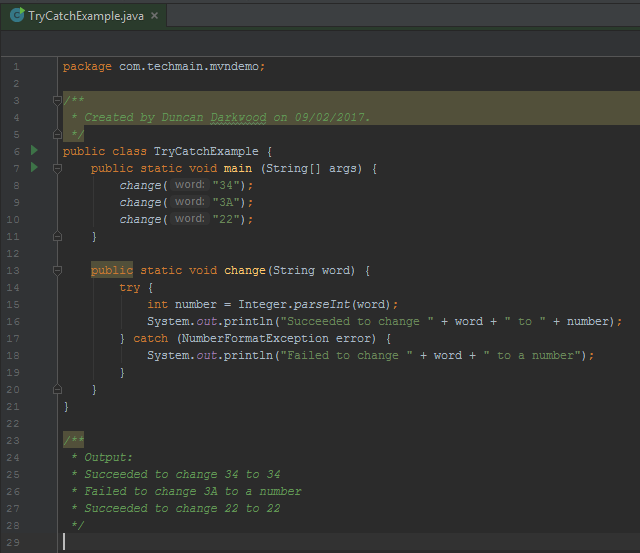
Hier nog een 20 minuten video die aanvullende en uitgebreide informatie geeft op het gebruik van exceptions. <https://www.youtube.com/watch?v=R86ObiKhMNc>

# 

# Try/Catch

Het idee van try/catch is vrij simpel. Je doet ongeveer hetzelfde als in een if/else structuur, maar dan met try/catch. Je probeert iets in try, en als dat niet werkt dan ga je naar catch. Wanneer er in try een error komt, wordt er in catch iets uitgevoerd, indeed er in try geen error komt, dan wordt er niets met catch gedaan.

Hieronder een voorbeeld van een try/catch structuur



Zoals te zien is, is poging 2, het veranderen van 3A naar een nummer, niet gelukt. Hierbij is het programma het catch stuk ingeslagen.

Als laatste, na een catch kun je een finally plaatsen. Alles wat er in de finally staat wordt ten alle tijden uitgevoerd, of de try is gelukt of niet.



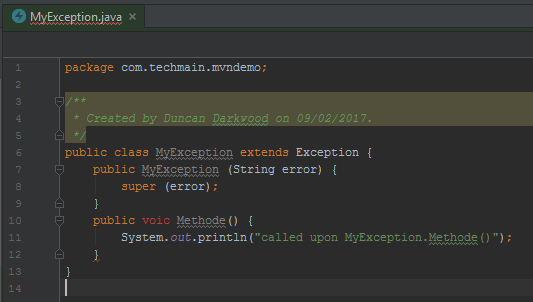
Hierboven is een try te zien met 2 mogelijke catches. Daarna is de finally te zien.

Wat hier het nut van is is dat je aan het eind altijd de output -\_-\_-\_-\_- hebt in de console (in dit voorbeeld), zonder dat het uitmaakt wat ervoor gebeurd is.

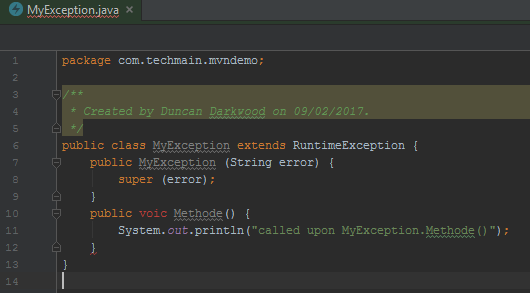
# Defineren

Voor het definiëren van een eigen exception moet er eerst bepaald worden of het een checked, of unchecked exception variant is. Dit moet omdat elke eigen exception een class is die extend naar een bestaande exception class. “Exception” voor checked exceptions en “RuntimeException” voor unchecked exceptions.

Hieronder een voorbeeld voor een checked exception

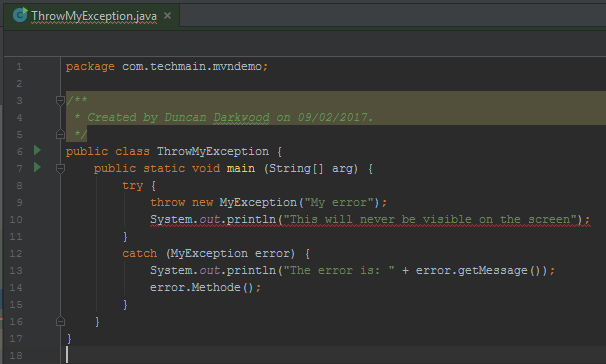


Hieronder een voorbeeld van een unchecked exception



# Throws/Throw

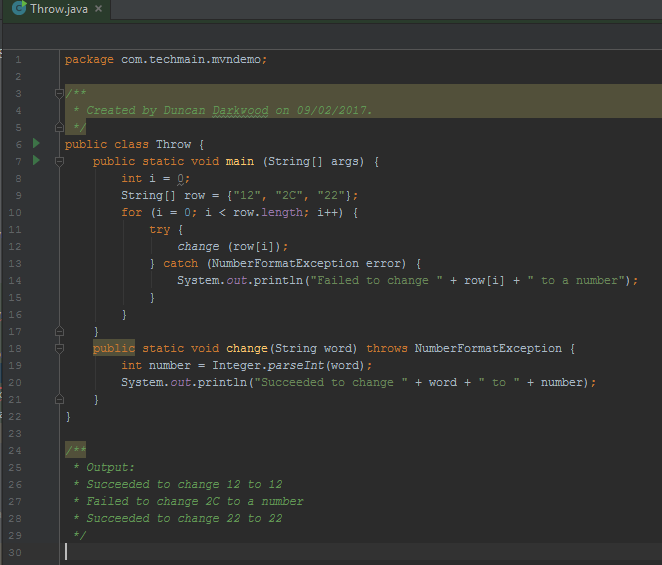
Met Throws en Throw worden er exceptions gegooid. Dit is om de gebruiker te informeren over een exception die is opgetreden en moet worden afgehandeld door de aanroeper.

Met als voorbeeld van ons eigen gedefineerde Exceptions kan Throw worden gebruikt.

Hier is te zien dat door middel van een try/catch de zelf gedefinieerde exception genaamd “MyException” wordt gegooid.

Er kan ook gebruik gemaakt worden van Throws, een keyword dat wordt gebruikt in een methode. Deze methode kan in de try worden aangeroepen, en geeft een specifieke exception terug in de catch die dan wordt afgehandeld.

Dit ziet er ongeveer zo uit



Zoals in dit voorbeeld te zien is, wordt er in try een methode aangeroepen genaamd “change”. Als er in deze methode een exception voorkomt, dan gooit (“throws”) hij het terug in de vorm van een NumberFormatException. Deze wordt bij catch dan weer uitgelezen en uitgevoerd.

Het verschil tussen throw en throws wordt op deze pagina uitgebreid uitgelegt http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/

In je App.java zet je dit

**package** com.techmain.mvndemo;

**import** java.util.InputMismatchException;

**import** java.util.Scanner;

*/\*\**

*\* Hello world!*

*\*/*

**public class** App {

**public static void** main(String[] args) {

**int** i = 0;

**boolean** done = **false**;

**do** {

**try** {

Scanner input = **new** Scanner(System.***in***);

System.***out***.println(**"enter an integer under or equal to 100"**);

i = input.nextInt();

**if** (i > 100) {

**throw new** intOutOfRangeException();

}

System.***out***.printf(**"you entered %d\n"**, i);

} **catch** (InputMismatchException e) {

System.***out***.println(**"That's not an integer, try again."**);

} **catch** (intOutOfRangeException e) {

System.***out***.println(**"That integer is too high, try again."**);

}

**finally** {

System.***out***.println(**"-\_-\_-\_-\_-"**);

}

} **while** (!done) ;

}

}

In je intOutOfRangeException class zet je

**package** com.techmain.mvndemo;

*/\*\**

*\* Created by allard on 09/02/2017.*

*\*/*

**public class** intOutOfRangeException **extends** Exception{

}

**User story 3: Unit testing**

**Beschrijving:**

*As a student, I need to learn unit testing so I can prove that my software matches the requirements.*

Unit Testing

Unit Testing is het testen van individuele stukken code. Dit is meestal het kleinste testbare stukje. In Procedural Programming kan dat een hele module zijn, maar vaker een functie of procedure. In object-oriented programming, een unit is vaak een hele interface, zoals een class, maar het kan ook een methode zijn.

Unit Testing wordt gebruikt om te zien of een stuk code daadwerkelijk doet en werkt zoals bedoelt is.

Hier wordt het nogmaals duidelijk uitgelegd:

<https://codeutopia.net/blog/2015/03/01/unit-testing-tdd-and-bdd/>

# Test Driven Development

Test Driven Development, ookwel TDD, is een process voor wanneer je je tests schrijft en runt. Met gebruik van TDD kan je een heel hoog test-coverage quota halen.

TDD wordt gedaan in de volgende stappen:

1. Start met het schrijven van een test
2. Run de test en alle andere tests die je hebt. Hier zou de nieuwe test moeten falen. Als dat niet het geval is zit er waarschijnlijk een bug in de test.
3. Schrijf het minimum aantal code dat nodig is om de test succesvol te laten wezen.
4. Run de test en check of de nieuwe test succesvol is. Zo ja: ga door naar stap 5. Zo nee: los het probleem op en voer daarna stap 4 uit.
5. Optimaliseer de code en check of test nog succesvol verloopt.
6. Herhaal vanaf stap 1.

Dit is in principe ook de red-green-refactor-cycle.

Hier een kata om dit te oefenen

<http://osherove.com/tdd-kata-1/>

**Unit testing via IntelliJ**

Als je in je IntelliJ bestand klassen of methodes met @Test markeert en je de nieuwste versie van JUnit in bezit hebt dan worden deze in de testfase van maven uitgevoerd. Ook worden ze uitgevoerd in IntelliJ als je op run whole project drukt of als je het testbestand runt.

Hier is wat code als voorbeeld voor het testen van een erg simpel stukje code (gejat van Déjan, bedankt).

|  |
| --- |
| package com.oose;  public class App {  public static void main(String[] args) {  System.out.println(getHelloWorld());  }   public static String getHelloWorld() {  return "Hello World";  } } |

Als je dit zou packagen en runnen dan zou het hello world printen.

Dit wil je natuurlijk wel testen voordat je het packaged zodat je niet tijd verspild in het geval dat het niet werkt en in plaats van hello world misschien wel goodbye world print.

Om dit te testen maak je een extra bestand aan in IntelliJ (of je pakt gewoon de automatisch gemaakte appTest). ←- volgens mij moet je per klasse een nieuwe testklasse maken. Bijvoorbeeld voor de class Calculator, maak je een testklasse CalculatorTest. Dit dan om meer structuur en duidelijk te krijgen in je code.

In die appTest doe je dan weer een stukje code schrijven:

|  |
| --- |
| package com.oose;  import org.junit.Assert; import org.junit.Test;  import static org.junit.Assert.assertTrue;  public class AppTest {  int value1 = 2;  int value2 = 3;  @Test  public void testPrintHelloWorld() {  Assert.assertEquals(App.getHelloWorld(), "Hello World");  }  @Test  public void testAdd(){  int result = value1 + value2;  assertTrue(result == 5);  } } |

In dit stukje code kijk je eerst of de methode getHelloWorld wel “Hello World” uitprint. Als de tekst exact overeenkomt met wat je bij assertequals aangeeft dan is de test succesvol uitgevoerd.

In het tweede stukje voeren we de test uit of 2 variabelen samen 5 zijn dit doen we hier met asserttrue. Als je meer wilt weten over assert dan lees deze link <https://github.com/junit-team/junit4/wiki/Assertions>

**Code coverage**

Code coverage is een afmeting om te laten zien hoeveel code van jou getest is. Dit word meestal aangeduid met een procentuele waarde, waarbij er gestreefd wordt naar 100% code coverage zodat je weet dat al jouw code getest is.

Om te kijken of je code coverage wel goed is in IntelliJ druk je op run (Alt+U) en dan op “run ‘appTest’ with coverage”. Dan laat hij zien hoeveel % van elke class, method en line gedekt is.

Dit knopje maakt een coverage rapport: 

Als je dan dubbelklikt op de coverage app dan gaat het kijken welke files specifiek gedekt zijn in plaats van het hele project. Dus dan zie je bijvoorbeeld dat App.java helemaal perfect is maar dat je andere klassen vol zitten met fouten.

# In unit tests checken of de juiste exceptions op het juiste moment komt

Je kunt op verschillende manieren testen of er een exceptie gegeven wordt door een functie:

Vanaf JUnit 4.7 (uitgebracht in 2009) kun je gebruik maken van de ExpectedException, waarop je dan weer de expect methode op kan aanroepen.

|  |
| --- |
| public class EenTest {  @Rule  public final ExpectedException exception = ExpectedException.none();   @Test  public void doIetsDatIndexOutOfBoundsExceptionThrowt() {  EenAndereKlasse eenKlasse = new EenAndereKlasse();  exception.expect(IndexOutOfBoundsException.class);  eenKlasse.doStuff();  } } |

Als je liever een eerdere versie van JUnit 4 wilt gebruiken kun je de verwachte waarde invullen in het Test keyword, zoals hieronder gebeurd:

|  |
| --- |
| @Test(expected = IndexOutOfBoundsException.class) public void testIndexOutOfBoundsException () {  ArrayList emptyList = new ArrayList();  Object o = emptyList.get(0); } |

Of als je helemaal niet gebruik wilt maken van JUnit om jouw code te testen, dan kan je natuurlijk een normale try-catch gebruiken.

**JUnit test framework** is wat we gebruiken:

Dit betekent dat we even deze site hier moeten lezen:

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

Een testcase is een aantal tests in een groepje dat gebruik maakt van dezelfde setup en teardown. Dit betekent dat je tests krijgt die makkelijk te herhalen zijn met dezelfde situaties die achter elkaar uitgevoerd kunnen worden.

Je kan het voorstellen als een testcase voor als admin inloggen en een voor als gebruiker inloggen. Het enige dat verschilt in de setup is dat je andere inloggegevens/methode gebruikt, maar in het programma zelf kan de admin veel meer. Je kan dan heel veel tests schrijven die allemaal de admin gegevens gebruiken, of je kan een keer in de setup de gegevens neerzetten en dan kan je die bij elke test in die testcase gebruiken.

**User story 4: Threading**

**Beschrijving:**

*As a student, I need to learn the Java Thread API so I can make applications run smoothly.*

Een thread is een set sequentiele stappen die worden uitgevoerd door een rekenkern van het systeem.

Threading is simple gezegd een software draad opdelen in meerdere delen voor de processorkern(en) zodat deze sneller opgelost kunnen worden.

Met een multicore processor wordt dit zo gedaan door multithreading. Hierdoor wordt het process opgedeeld door meerdere kernen, deze voeren dan paralell de instructies uit(een function call, een loop etc). Omdat het process wordt opgedeeld in meerdere delen en tegelijk uitgevoerd wordt is het process sneller af te handelen door de processor.

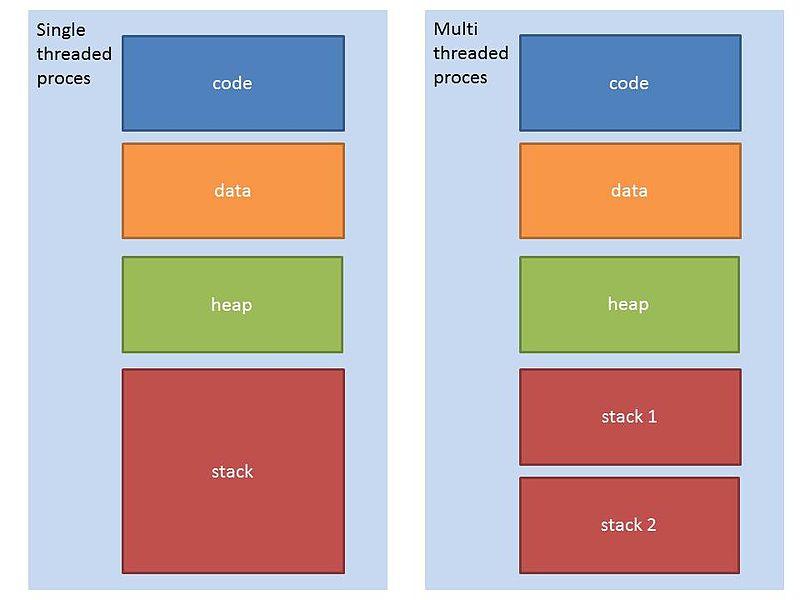
Op een CPU met een enkele processor wordt mogelijk gebruik van threading door middel van time slicing. Dit houdt in dat de processor wisselt tussen software threads. Hierdoor doet nog steeds de enkele processor al het werkt, maar worden er meerdere delen van de thread gedaan waardoor het lijkt alsof er processen in parallel lopen terwijl dit niet zo is.

Voorbeeld van single core en multithreading (vertaald van een reddit post maar zeer duidelijk)

Voor nog een simple voorbeeld, Stel je voor dat je een grote maaltijd maakt voor thanksgiving. Elk recept is een lijst instructies net zoals een thread en de persoon die het kookt is net een processor. Je zou de kalkoen alvast in de oven kunnen stoppen en dan beginnen aan een taart. Dan daarmee stoppen om aardappelen te pureren. Je werkt maar tegelijk aan één recept maar maakt wel voortgang op alle recepten omdat je steeds tussen recepten wisselt. Als je een tweede persoon zou toevoegen kun je aan twee recepten tegelijk werken, net alsof je twee processoren hebt.

Uit dit voorbeeld is duidelijk te zien hoe een tweede kern het werk kan versnellen door de recepten te verdelen door twee personen.

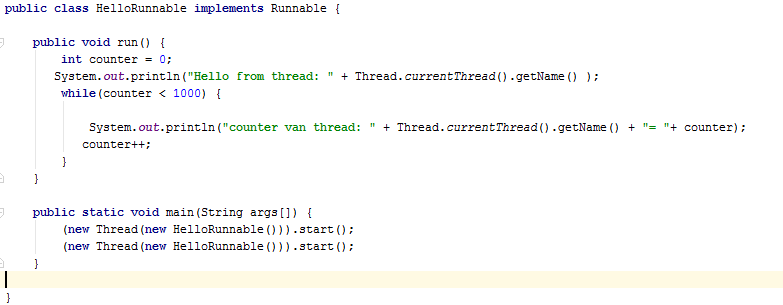
In figuur 1 is te zien hoe de threads verdeeld worden.



Wat hieruit blijkt is dat alleen de stack verdeeld wordt en de rest niet. Dit heeft als gevolg dat er geen contextwijzingen zijn tussen de threads en dit leidt tot een snellere oplossing.

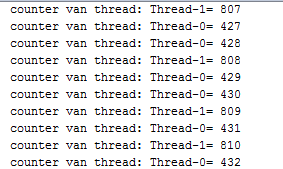
# Threads implementeren met Runnable en Thread.

Als je een programma wilt laten werken met threads, met je de klasse de interface Runnable laten implementeren.



Nadat je de klasse hebt aangemaakt moet je daarin de methode run() aanpassen naar de uitvoer van wat je wilt dat er gebeurt. run() MOET je implementeren, dit is de methode die door Thread.start wordt aangeroepen.

Daarna maak je een nieuwe instantie van de door java geleverde klasse Thread en geef je als parameter een nieuwe instantie van de klasse die runnable implementeert. Daardoor worden er twee threads aangemaakt en die



In het voorbeeld is te zien dat de tellers steeds apart optellen, dit gebeurt allebei op een aparte thread ze zijn dus allebei “tegelijk” bezig met het runnen van hun deel van het programma.

Er is een andere manier omdat te doen die niet bij onze user story aan bod komt, mochten jullie dit willen zal ik dit ook nog uitwerken. Runnable implementeren is meest gebruikte manier.

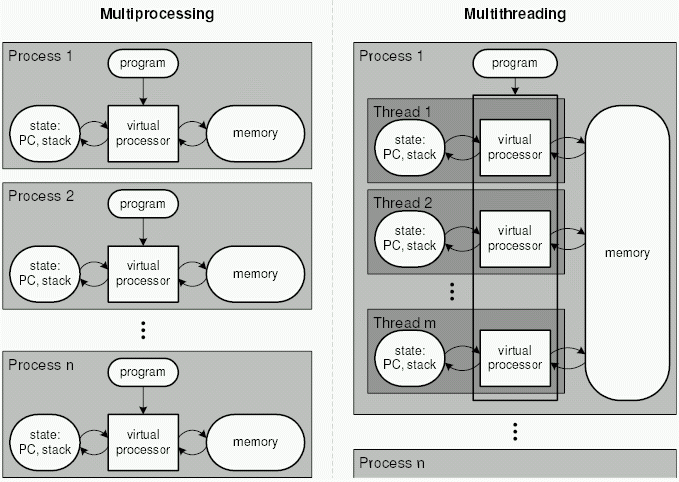
# Verschil tussen multi-threading en multi-processing

Het verschil tussen multithreading en multiprocessing is dat processes per proces een ander plekje in het geheugen innemen, waardoor er niet zo gemakkelijk onderlinge communicatie mogelijk is zoals bij multithreading het geval is. In het voordeel van multi-processing heb je dat er meerdere processen kunnen gemaakt worden over meerdere CPU’s, terwijl je bij multithreading vast zit op één CPU (maar wel meerdere cores op diezelfde CPU).

Verder zou je kunnen beargumenteren dat je met meerdere processen wat immuner bent tegen fouten en crashes: als je bijvoorbeeld twintig processen hebt en in ééntje ontstaat een fatale fout, dan crasht alleen die ene waardoor je nog negentien processen overhoudt. Als je dit nou met multithreading vergelijkt: alle threads zitten op één proces, dus als er een fatale fout gebeurd bij een thread, dan crasht als gevolg dat ene proces.

Het nadeel hiervan is wel dat meerdere threads sneller zijn dan dan meerdere processen, omdat threads veel minder geheugen innemen en je hoogstwaarschijnlijk meer benodigdheden hebt voor processen. In het geval van Java maakt elk nieuwe proces een nieuwe Java Virtual Machine aan, wat ten koste gaat van bijvoorbeeld de opstarttijd, terwijl een thread dat niet doet.

*Onderstaande afbeelding laat in een snel opzicht het verschil zien tussen multiprocessing en multithreading.*



# Threading en de rol van de JVM, een besturingssysteem en de processor met zijn cores hierin spelen.

In het geval van multithreading wordt er eenzelfde Java Virtual Machine gebruikt, waardoor objecten en attributen met de threads gedeeld kan worden, terwijl de Java Virtual Machine maar één proces is. Als er binnenin die JVM gevraagd wordt om een andere thread te maken, dan wordt er één thread binnen die JVM gemaakt. Het besturingssysteem regelt dat deze threads op verschillende cores zitten waardoor de snelheid veel hoger ligt dan wanneer het allemaal op één core ligt.

# Een thread correct starten en stoppen en de beperkingen van de API bij het stoppen.

## Starten

Om een thread correct te starten moet je gebruik maken van de start methode. Deze methode zit in de Thread class of in de Runnable interface, die je dus moet uitbreiden of implementeren zoals hieronder gebeurd:

|  |
| --- |
| public KlasseNaam extends Thread |

|  |
| --- |
| public KlasseNaam implements Runnable |

De aanbevolen manier is door de Runnable interface te implementeren, al kan dit er wel voor zorgen dat de code wat uitgebreider word: als je een Thread extend kan je namelijk gewoon start op een instantie van die klasse aanroepen, maar als je Runnable implementeert moet je een nieuwe instantie maken van de Thread klasse, met als argument jouw eigen klasse, en daarop moet je de start methode aanroepen. Waarschijnlijk is het wat makkelijker om te tonen in code:

|  |
| --- |
| new KlasseNaam().start() // KlasseNaam extends Thread |

vs.

|  |
| --- |
| new Thread(new KlasseNaam()).start() // KlasseNaam implements Runnable |

De run methode wordt daarna automatisch aangeroepen, en de thread is dus klaar met taken uitvoeren wanneer hij bij het eind is van de run methode.

## Stoppen

KlasseNaam.currentThread().interrupt(), en dan checken of KlasseNaam.currentThread().isInterrupted() of een while loop met een volatile variable (zoals running) zodat de functie eindigt

Het stoppen van de threads is niet zo eenvoudig, met verschillende manieren om te stoppen en verschillende beperkingen per methode. Je ‘stopt’ de thread namelijk door aan het eind van de run functie te komen, en gebaseerd op het doel van jouw applicatie kan het verschillen hoe je dat doet. Hoe je een thread kan stoppen is als volgt:

* Jouw run methode is aan het eind van de functie beland;
* Jouw run methode retourneert iets, en gebaseerd op jouw code kan er een situatie ontstaan waar je niets wil retourneren maar alsnog moet om de thread te stoppen;
* Je roept de Thread-methode ‘interrupt’ aan. Let wel op dat dit niet per se de thread stopt, want de ontwikkelaar kan zelf aangeven wat de thread moet doen. Je kan dan checken of jouw thread ge-interrupt is door de Thread-methode ‘isInterrupted’.
* Je creëert een variabele in jouw thread die volatile is, en checkt met bijvoorbeeld een while loop of die variabele true, en als dat zo is dan word een taak uitgevoerd. Volatile variabelen zijn variabelen die niet geoptimaliseerd worden door de Java compiler, en kan je gebruiken om aan te geven dat de waarde van de variabele aangepast gaat worden door meerdere threads. Dit is in mijn ogen de meest gebruikte manier om een thread te stoppen;

Je beëindigt het proces waar de thread in draait, alleen wil je dit natuurlijk niet in elke situatie doen.

# Kan het keyword synchronized correct inzetten om te voorkomen dat threads tegelijk een gedeelde resource benaderen.

Stel dat je Java-programma twee threads creëert, met beide als doel om van nul naar honderd te tellen en elke stap te melden aan de console. Zonder synchronized zou dat betekenen dat er in de console heel vaak niet van correct van nul naar honderd geteld wordt. Je zou dan bijvoorbeeld 1, 1, 2, 3, 2, 4, 3 zien.

Met behulp van synchronisatie kan maar één thread per keer de gesynchroniseerde methode aanroepen, waarna threads die die methode ook willen aanroepen moeten wachten tot de andere thread klaar is met die methode.

Het stukje tekst hierboven kan je als volgt uitwerken:

|  |
| --- |
| public class Example {  public static void main(String[] args) {  Counter counterA = new Counter();  Thread[] threads = new Thread[20];  // Maak de threads aan  for (int i = 0; i < threads.length; i++) {  threads[i] = new Thread(new CounterThread(counterA));  }  // Start de threads  for (int i = 0; i < threads.length; i++) {  threads[i].start();  }  } } |

|  |
| --- |
| public class CounterThread implements Runnable {  protected Counter counter = null;   public CounterThread(Counter counter) {  this.counter = counter;  }   public void run() {  for (int i = 0; i < 100; i++) {  this.counter.add(i);  try {  Thread.sleep(20);  } catch (InterruptedException e) {  System.out.println(e);  }  }  } } |

|  |
| --- |
| public class Counter {  long count = 0;   public synchronized void add(long value) {  this.count += value;  System.out.println("Toegevoegde hoeveelheid: " + value + " - Totale hoeveelheid: " + this.count);  } } |

Wanneer je de functie add in de klasse Counter niet gesynchroniseerd maakt, dan zou je uiteindelijk een verkeerde totale hoeveelheid hebben omdat alle threads tegelijkertijd proberen om de variabele count aan te passen. De correcte hoeveelheid is 99000. Zonder synchronized is de kans heel erg groot dat je lager dan dit getal uitkomt, want soms wordt de functie add door meerdere threads aangeroepen.

**User story 5: RMI**

**Beschrijving:**

*As a student, I need to learn the RMI-protocol so I can build distributed applications and know the concepts of distributed computing.*

**Acceptatiecriteria:**

* Wat is RMI?

**Uitleg/uitwerking:**

Remote Method Invocation is een vorm van Remote Procedure Call (RPC) en een standaard onderdeel van Java. RMI laat toe om applicaties te schrijven die methodes op andere Java virtuele machines (JVMs) kunnen uitvoeren.

Hiervoor is een referentie nodig naar het remote object. Dit kan mbv een naam-dienst (bootstrap) or een methode aan te roepen die een object referentie oplevert.

Om aanroepen van methoden tussen programma's mogelijk te maken is het volgende nodig:

- server moet bekend maken bij welke objecten methoden kunnen worden aangeroepen

- server moet bekend maken welke methoden dat zijn aanroepen en returnwaarden moeten in een serie bits vertaald kunnen worden (voor verzending) en vice versa

- concrete instanties van objecten die aan de serverkant aangesproken kunnen worden moeten zich bekend maken

Voorbeeld:

**public interface *ObjIf* extends Remote**

**{**

**public *returnwaarde methode*(*parameters*)**

**throws RemoteException;**

**// meer methoden**

**}**

* Kan een interface definieren voor een remote class gebruik makend van de API-onderdelen Remote en RemoteException

**Uitleg/uitwerking:**

De volgende code is een voorbeeld van een interface voor remote classes:

import java.rmi.Remote;

import java.rmi.RemoteException;

public interface Hello extends Remote {

String sayHello() throws RemoteException;

}

* Kan een class remote benaderbaar maken gebruik makend van de API-onderdelen UnicastRemoteObject en Java interfaces.

**Uitleg/uitwerking:**

De code om de class remote benaderbaar te maken gaat via de volgende code:

Op zichzelf heb je misschien niet veel aan deze code, maar in context (zie volgende

punt over RMI) moet het duidelijk zijn hoe de code gebruikt dient te worden.

Hello stub = (Hello) UnicastRemoteObject.*exportObject*(obj, 0);

// Bind the remote object's stub in the registry

Registry registry = LocateRegistry.*getRegistry*();

registry.bind("Hello", stub);

* Kan een server voor een remote class schrijven die via het RMI-protocol en de Registry aanroepen van een client kan ontvangen.

**Uitleg/uitwerking:**

Onderstaande code is een voorbeeld van een server aanmaken.

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {

public Server() { }

public String sayHello() {

return "Hello, world!";

}

public static void main(String args[]) {

try {

Server obj = new Server();

Hello stub = (Hello) UnicastRemoteObject.*exprmiortObject*(obj, 0);

// Bind the remote object's stub in the registry

Registry registry = LocateRegistry.*getRegistry*();

registry.bind("Hello", stub);

System.*err*.println("Server ready");

} catch (Exception e) {

System.*err*.println("Server exception: " + e.toString());

e.printStackTrace();

}

}

}

* Kan een client schrijven die via het RMI-protocol en de Registry methoden op een remote class aanroept.

**Uitleg/uitwerking:**

Zie de volgende code als voorbeeld voor een client:

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

public class Client {

private Client() {}

public static void main(String[] args) {

try {

Registry registry = LocateRegistry.*getRegistry*();

Hello stub = (Hello) registry.lookup("Hello");

String response = stub.sayHello();

System.*out*.println("response: " + response);

} catch (Exception e) {

System.*err*.println("Client exception: " + e.toString());

e.printStackTrace();

}

}

Belangrijke code hierin is het deel over de registry en de stub. Op deze manier

maak je als het ware connectie met de registry. Deze registry kent de beschikbare

methodes op de server. Deze methode is namelijk via een interface gebonden aan

de registry.

Nu kan dankzij de registry via de stub een methode op de server (een andere JVM)

aangeroepen worden.

* Weet het verschil tussen pass-by-value en pass-by-reference en kan het API-onderdeel Serializable inzetten voor pass-by-value.

Uitleg/uitwerking:

* Kent het concept van client-registry-server.

**Uitleg/uitwerking:**

Het idee hier is dat de registry een soort wegwijzer is. In de registry staat opgeslagen

welke methodes zich in welke class bevinden. Door deze registry te gebruiken op

een server kunnen methodes aan de registry gekoppeld worden. Een cliënt kan via

de registry weer methodes aanroepen van een server of andere JVM. Hiervoor moet

wel een stub aangemaakt worden, deze representeert de methodes uit andere JVM’s

in de huidige klasse.

* Hoeft geen rekening te houden met security (SecurityManager), codebases en classpaths.

**RMI introductie.**

RMI(Remote method invocation) is het aanroepen van een functie van een methode op een andere JVM(Java Virtual Machine) door gebruik van de registry(van Java niet te verwarren met de Windows Registry). Hierdoor kan een JVM dus methodes gebruiken van een andere JVM. De registry dient hierbij als wegwijzer. Dit kan lokaal op dezelfde computer en via het internet op een andere computer.

In het voorbeeld is een stappenplan te zien van RMI. Eerst start de server de registry en blijft ermee in contact. Dan pingt de cliënt de registry voor de locatie van de server, dan geeft de client zijn command (waarschijnlijk met een methode aanroep op de server). De server voert deze methode door de remote methode aanroep van de cliënt.

**RMI command line guide:**

Om RMI via de command line op te starten, een server aan te zetten en een class aan te roepen kun je deze korte handleiding volgens.

Ten eerste moet je project gebuild worden. Navigeer in de command line naar de directory waar je Maven project staat. Run vervolgens een command als mvn clean package.

Het is nu tijd om de rmiregistry te starten.

Dit doe je door in de command line rmiregistry (macOS) of start rmiregistry (Windows) in te voeren.

Een voorbeeld van OSX code: rmiregistry &

Nu is het tijd om de server te starten. Doe dat met deze code:

java -classpath . -Djava.rmi.server.codebase=file:./ Server

Het is hierbij belangrijk dat de working directory wel de map is waarin de bestanden staan. In mijn geval is dat bijvoorbeeld: dropbox/oose/projects/maven/rmi-helloWorld/target/classes

Open nu een nieuw command-line venster. In dit venster gebruik je de volgende code om een cliënt te starten: java -classpath . Client

Wanneer de applicatie juist is gebouwd, en zoals het voorbeeld een response moet geven, dan krijg je als het goed is het volgende: response: Hello, world!

Notitites van allard: download het meron ding

En in system environment variable moet je "C:\Program Files\Java\jdk1.8.0\_121\bin" toevoegen aan je path

Of een andere java versie als je dat hebt

Maar iniedergeval de bin van je java map

#windowsproblems

#windowsSolutions