

Java

Les 8

jeroen.devos01@ap.be

Lesdoelen

- Multithreading
- Javadoc
- Software architectuur
- Herhaling hibernate



Advanced Java™

Theorie – Threading

Threading is een programmeertechniek waarbij meerdere taken tegelijk uitgevoerd kunnen worden.

Een nieuwe thread kan dus in parallel worden uitgevoerd met andere code.

Je software heeft minstens één thread = main thread.

Voordelen:

- Sneller berekeningen maken
- Lange / moeilijke taken in de achtergrond uitvoeren terwijl het programma blijft werken

Threading voorbeeld

Case :

Neem een lijst met 400 getallen en verdubbel elk getal in de lijst.

Optie 1 : zonder thread -> uitvoering sequentieel

Threading voorbeeld

Case :

Neem een lijst met 400 getallen en verdubbel elk getal in de lijst.

Optie 1 : zonder thread -> uitvoering sequentieel

Optie 2 : met 2 threads

- Thread 1 doet 1 tot 200
- Thread 2 doet 201 tot 400

Threading voorbeeld

```
Thread t1 = new Thread(new Doubler(list1));  
Thread t2 = new Thread(new Doubler(list2));  
t1.start();  
t2.start();  
System.out.println(list1);  
System.out.println(list2);
```

- Aanmaken new Thread
 - Opgelet class in Thread moet Runnable zijn
- Thread opstarten

Threading voorbeeld – 4 threads

Quad Core Machine

Input Array

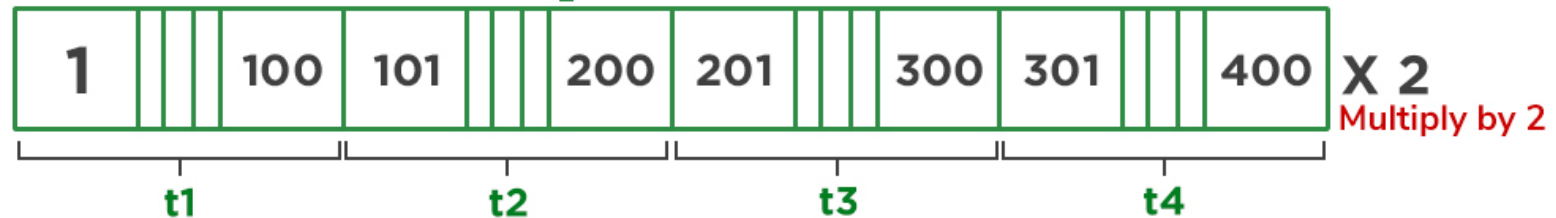


Output Array



4 sec ↓ without threads QuadCore Machine

Input Array



1 sec ↑ with threads QuadCore Machine

Threading limitaties

Blijf je snelheidswinst maken als je meer threads aanmaakt?

Bijvoorbeeld 8 thread of 16 threads?

Threading limitations

Blijf je snelheidswinst maken als je meer threads aanmaakt?

Bijvoorbeeld 8 thread of 16 threads?

NEEN!

Je bent gelimiteerd tot het aantal multicore processoren in je machine!

Elke thread heeft zijn eigen core nodig.

Heb je meer threads dan cores, dan wordt er heel snel geswitched tussen cores en lijkt het maar alsof je parallel aan het rekenen bent.

Class Thread

- new Thread(Runnable);
- new Thread(Runnable, String); <- threadname
- Methods
 - start()
 - isAlive()
 - interrupt()
 - isInterrupted()
 - sleep(timeout)
 - wait()
 - notify()
 - wait(timeout)



Runnable interface

```
public class Doubler implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("This code is running in a thread.");  
    }  
}
```

- Thread kan enkel classes die Runnable zijn uitvoeren
- Method run() : deze wordt door de thread uitgevoerd

start() & isAlive()

```
Thread t1 = new Thread(new Doubler(list1));  
t1.start();  
while(t1.isAlive()) {  
    System.out.println("Thread is running...");  
}  
System.out.println(list1);
```

- Gebruik while loop met controle op isAlive() om te wachten tot de thread klaar is

interrupt() en isInterrupted()

```
public class Doubler extends Thread {  
  
    @Override  
    public void run() {  
        if(!isInterrupted()) {  
            System.out.println("This code is running in a thread.");  
        }  
    }  
}
```

- Thread moet afgewerkt worden
- Stoppen v/e Thread -> interrupt() method
- Voorzie een controle op isInterrupted() in de run method

Sleep(timeout)

```
public class Doubler extends Thread {  
  
    @Override  
    public void run() {  
        try {  
            sleep(1000);  
        } catch (InterruptedException e) {  
        }  
    }  
}
```

- Method `sleep(timeout)` legt de thread stil voor een aantal milliseconden.
- Tijdens de wachttijd houdt de thread een lock op zijn object.

Sleep(timeout)

```
public class Doubler implements Runnable {  
  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
        }  
    }  
}
```

- In geval van Runnable kan je Thread.sleep gebruiken

wait() & notify()

```
Thread t1 = new Thread(new Doubler(list1));  
t1.start();  
t1.wait();
```

```
//in een andere thread ...  
t1.notify();
```

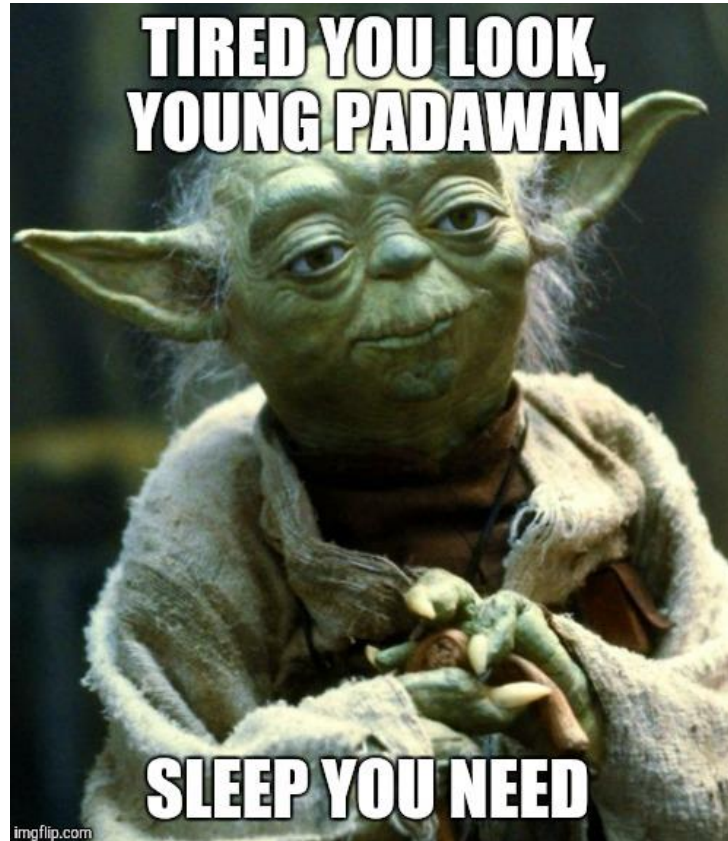
- Method wait() legt de thread stil.
- Thread blijft wachten tot in een **andere** thread de notify() method wordt gebruikt.
- Tijdens het wachten worden alle objecten vrijgegeven (geen lock).

wait(timeout)

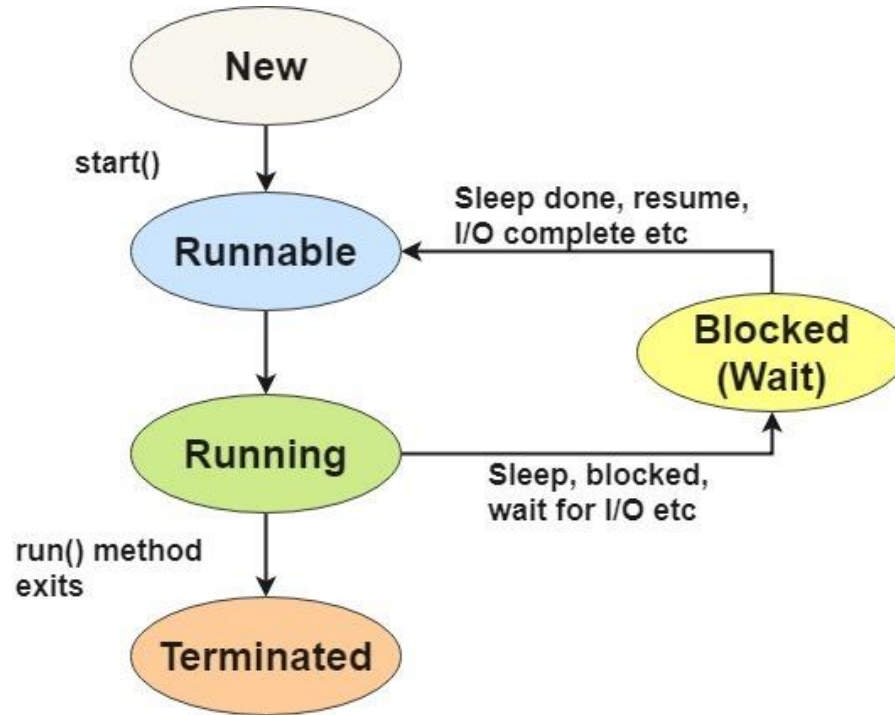
```
Thread t1 = new Thread(new Doubler(list1));  
t1.start();  
t1.wait(1000);
```

- Method wait(timeout) legt de thread stil voor een aantal milliseconden.
- Thread blijft wachten tot de timeout verlopen is **OF** tot in een andere thread de notify() method wordt gebruikt.

Sleep or wait?



Thread lifecycle



Race conditions

Multithreaded programming



Race condition

Door gebruik van threads is de volgorde van uitvoering van je code niet meer gegarandeerd.

Hierdoor kunnen onverwachte situaties ontstaan.

Race condition demo

```
public class Counter implements Runnable {  
    private int shared = 0;  
  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(10);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        shared++;  
        System.out.println("After incr " + shared);  
    }  
}
```

```
for (int i = 0; i < 10; i++) {  
    Counter counter = new Counter();  
    Thread t1 = new Thread(counter, "T1");  
    Thread t2 = new Thread(counter, "T2");  
    t1.start();  
    t2.start();  
    while(t1.isAlive() || t2.isAlive()) {  
        //wachten tot de threads klaar zijn  
    }  
    System.out.println("*****");  
}
```

Race condition demo - resultaten

Thread 1	Thread 2		value
			0
read value		←	0
value++			0
write value		→	1
	read value	←	1
	value++		1
	write value	→	2



Thread 1	Thread 2		value
			0
read value		←	0
	read value	←	0
value++			0
	value++		0
write value		→	1
	write value	→	1



4, 5, 6 before 1, 2, 3 ?

Synchronization, you have not used

Meme-Generator.com

Race condition - oplossing

```
public void run() {  
    synchronized (this) {  
        try {  
            Thread.sleep(10);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        shared++;  
        System.out.println("After incr " + shared);  
    }  
}
```

- Op code block of op method
- Alle threads zien dit stuk code.
- Code kan slechts één maal tegelijkertijd uitgevoerd worden.

Race condition – meer oplossingen

Er zijn nog manieren om met race condition om te gaan:

Volatile variables en Atomic variables



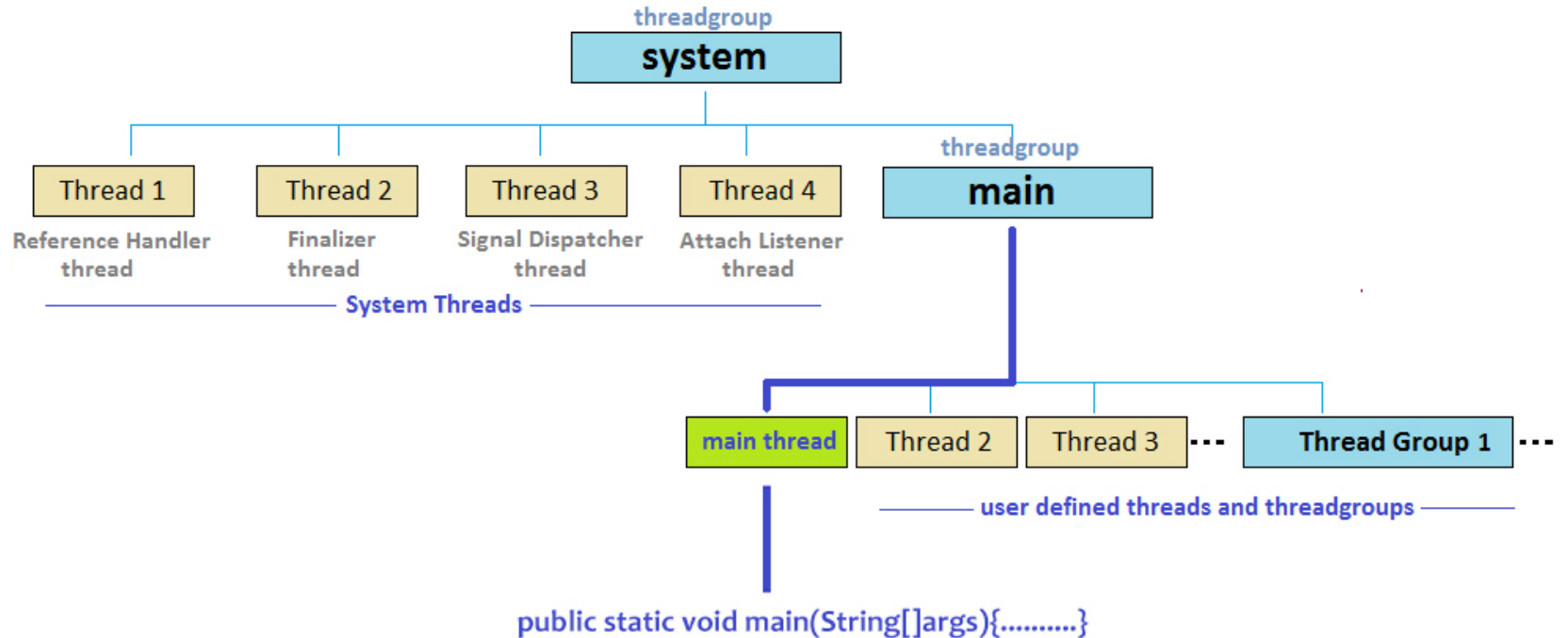
Types threads

Java heeft twee soorten threads:

- User threads
 - Main is de eerste user thread in een applicatie
 - Een java applicatie komt ten einde wanneer alle user threads op zijn.
- Deamon threads
 - Service provider voor de user threads
 - Bijvoorbeeld garbage collector

Types threads

Thread Group Hierarchy



Thread safe classes

- Classes zijn "thread-safe" wanneer ze zodanig zijn geïmplementeerd dat er geen race conditions kunnen optreden.
- Classes die niet thread-safe zijn, mogen dus niet simultaan worden gebruikt door verschillende threads door de geziene risico's !
- Indien je zelf je classes thread safe moet maken: gebruik dan:
 - Synchronized methodes en/of
 - Volatile variables en/of
 - Atomic variables

Thread safe collections

- Collections zijn standaard NIET thread-safe !
 - Maar sommige wel: Stack, Vector, Hashtable (onder andere)
- Er zijn statische 'synchronized'-methods (wrappers) om deze om te zetten naar thread-safe objecten:
 - `static Collection synchronizedCollection(Collection c)`
 - `static List synchronizedList(List list)`
 - `static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)`
 - `static Set synchronizedSet(Set s)`
 - `static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)`
 - `static SortedSet synchronizedSortedSet(SortedSet s)`
- Dit lockt tijdens gebruik de hele collectie. Er kan dus maar 1 thread tegelijkertijd de lijst bewerken.



**THREAD-SAFE
COLLECTIONS**

Thread safe collections - voorbeeld

```
List<String> syncList = Collections.synchronizedList(Arrays.asList("a", "b", "c"));
List<String> uppercaseList = new ArrayList<>();

public void run() {
    synchronized (syncList) {
        syncList.forEach(item -> uppercaseList.add(item.toUpperCase()));
    }
}
```

/**

* Easy

* Javadoc

*/

Wat is ...

Javadoc = Java documentation

Documentatie van je code wordt mee in de code geschreven.
Nadien wordt deze documentatie gegenereerd naar html.

Bv: <https://docs.oracle.com/en/java/javase/21/docs/api/index.html>

Javadoc toevoegen

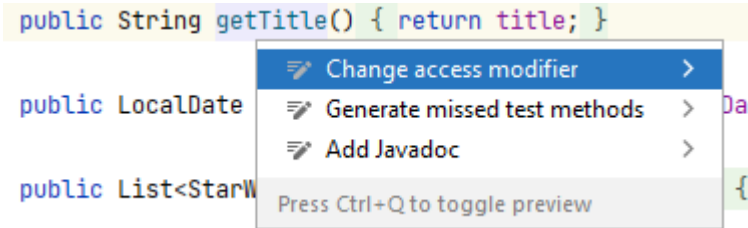
```
/**  
 * Een film van de Star Wars reeks.  
 */  
public class StarWarsMovie
```

- Plaatsing bij een class, methode of attribuut.
- Begint met **/**** en eindigt met ***/**
- Opgelet : niet verwarren met commentaar : begint met **/*** en eindigt met ***/**

Javadoc toevoegen

```
/**  
 * Een film van de Star Wars reeks.  
 */  
public class StarWarsMovie
```

- IntelliJ shortcut :
 - cursor op class/method/attribute
 - ALT+ENTER
 - Selecteer 'Add Javadoc'



Javadoc opmaak : html

```
/**  
 * Een film van de <b>Star Wars</b> reeks.<br>  
 * <u>Must-see</u> voor iedereen!<br>  
 * Meer info <a href="https://www.imdb.com/star-wars/">IMDB</a>  
 */  
public class StarWarsMovie
```

- Opmaak van de Javadoc via html tags.
- Meeste tags zijn toegelaten...

Javadoc metadata

```
/**  
 * Een film van de Star Wars reeks.  
 * @author Jeroen De Vos  
 * @version 1.0  
 */  
public class StarWarsMovie
```

- Metadata → annotations !
- Syntax : annotation + spatie + eigen tekst

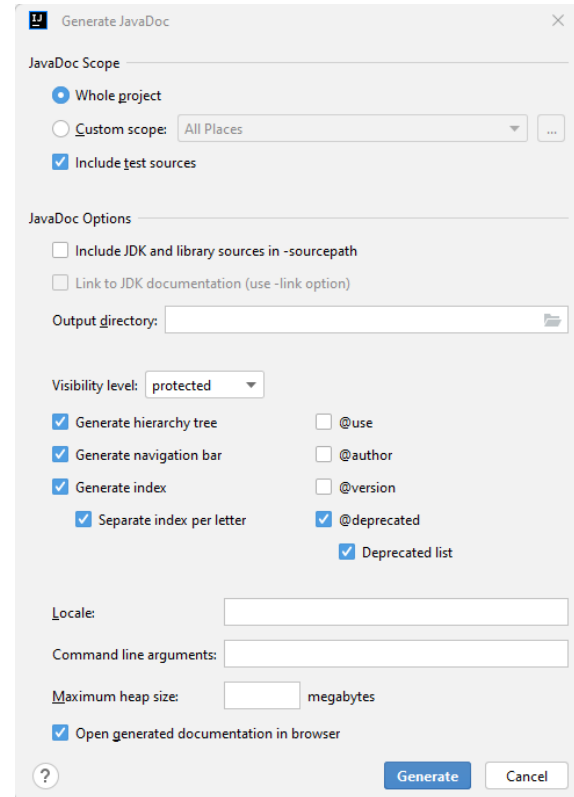
Javadoc metadata : method

```
/**  
 * Controleert of het character de vader is van het input character.  
 * @param character : het mogelijke kind  
 * @return true indien het character de vader is, false indien het niet de vader is  
 * @throws StarWarsFatherException : wanneer het input character dezelfde is als het character  
 */  
public boolean isFather(StarWarsCharacter character) throws StarWarsFatherException
```

- Methodes hebben specifieke annotations:
 - **@param** : beschrijft elke input parameter.
 - **@return** : beschrijft wat er terug komt uit de method.
 - **@throws** : beschrijft wanneer de foutmelding optreedt.

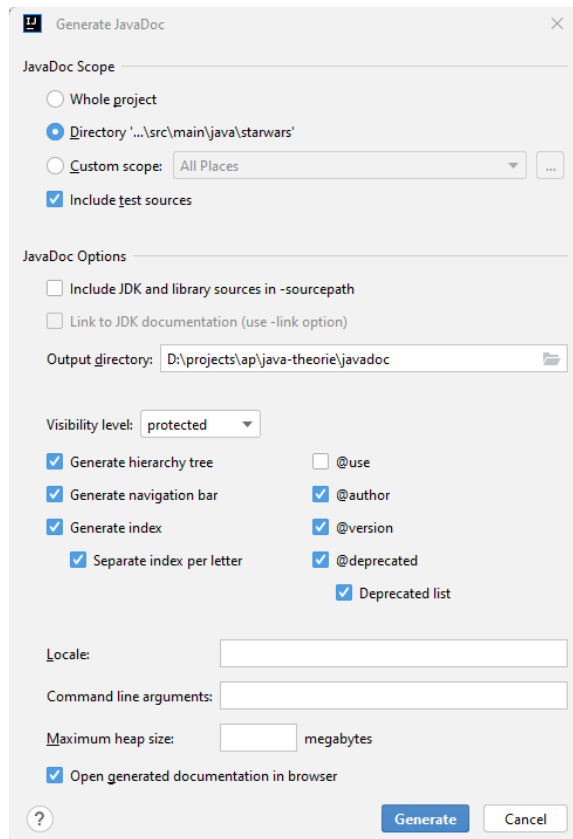
Javadoc genereren (IntelliJ)

- Tools > Generate Javadoc

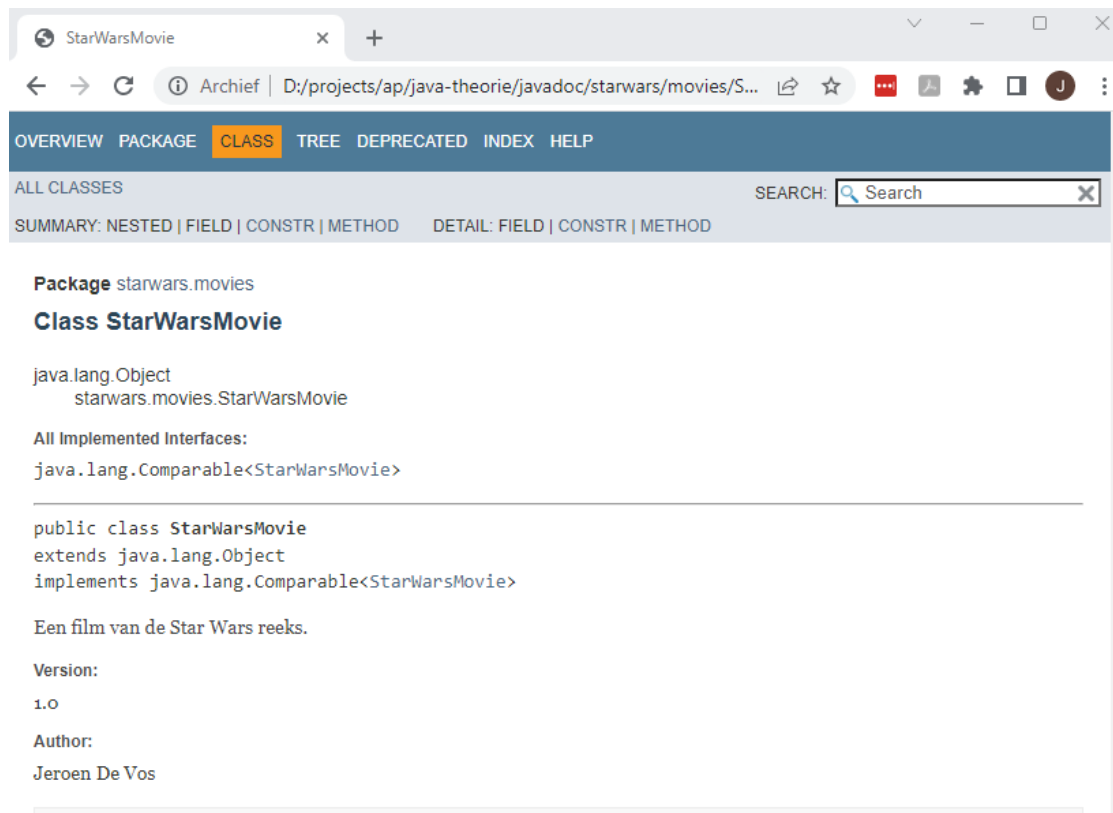


Javadoc genereren (IntelliJ)

- Scope :
 - Kies welke bestanden
 - Selecteer eventueel package
- Options :
 - Vul de output in
 - Selecteer @author en @version



Javadoc genereren (IntelliJ)



The screenshot shows a web browser window with the address bar displaying the path `D:/projects/ap/java-theorie/javadoc/starwars/movies/S...`. The browser's navigation bar includes tabs for 'Overview', 'Package', 'Class' (which is selected and highlighted in orange), 'Tree', 'Deprecated', 'Index', and 'Help'. Below this, there is a search bar with the placeholder text 'SEARCH: Search'. The main content area displays the Javadoc for the `StarWarsMovie` class. It starts with the package `starwars.movies` and the class name `Class StarWarsMovie`. The class is shown to extend `java.lang.Object` and implement `java.lang.Comparable<StarWarsMovie>`. The Javadoc includes a description: 'Een film van de Star Wars reeks.' and metadata for 'Version: 1.0' and 'Author: Jeroen De Vos'.

StarWarsMovie

Archief | D:/projects/ap/java-theorie/javadoc/starwars/movies/S...

OVERVIEW PACKAGE **CLASS** TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package starwars.movies

Class StarWarsMovie

java.lang.Object
starwars.movies.StarWarsMovie

All Implemented Interfaces:
java.lang.Comparable<StarWarsMovie>

```
public class StarWarsMovie
extends java.lang.Object
implements java.lang.Comparable<StarWarsMovie>
```

Een film van de Star Wars reeks.

Version:
1.0

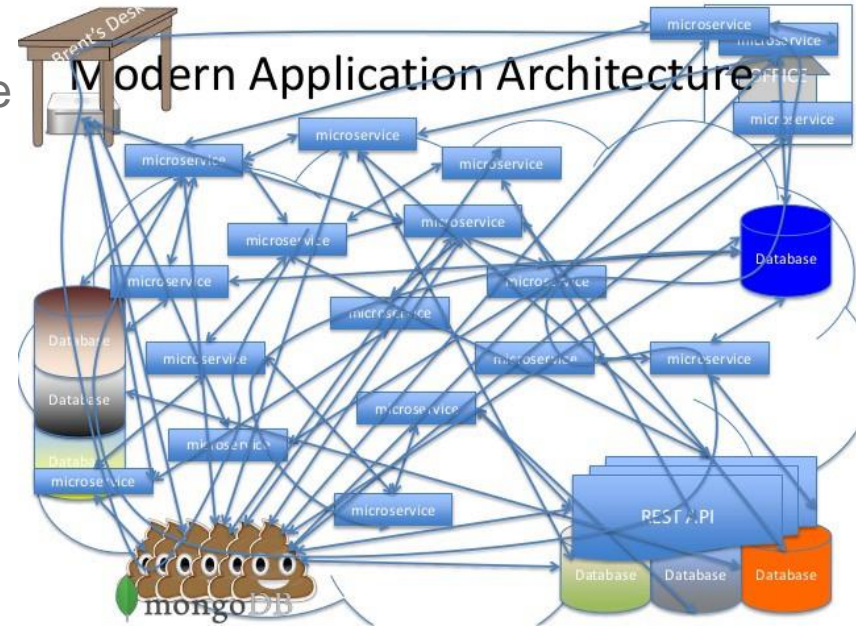
Author:
Jeroen De Vos

Software architecture



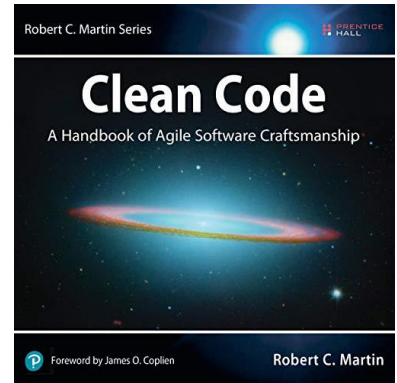
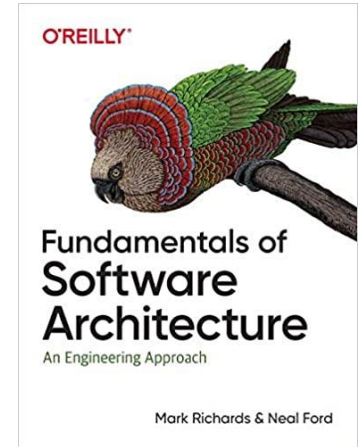
Wat is ...

- Software architecture dient om grote toepassingen beheersbaar en onderhoudbaar te maken.
- Zonder een duidelijke plan kan software al snel 'spaghetti' worden.
- Software architect = beroep !

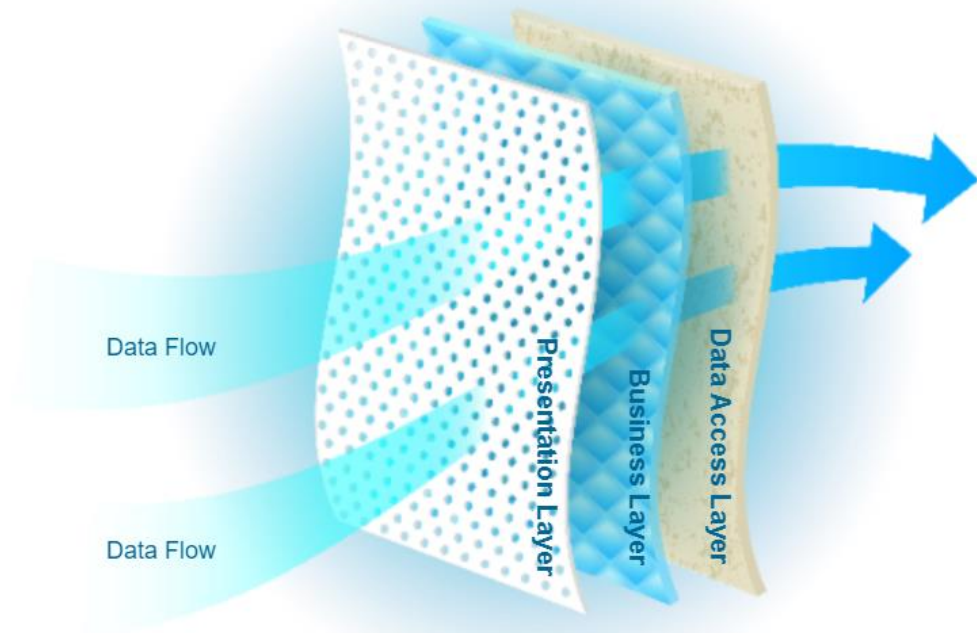


Software architecture voorbeelden

- Layered
- Command Query Responsibility Segregation (CQRS)
- Model – View – Controller (MVC)
- Microservices
- Strangler



Layered architecture



Wat is ...

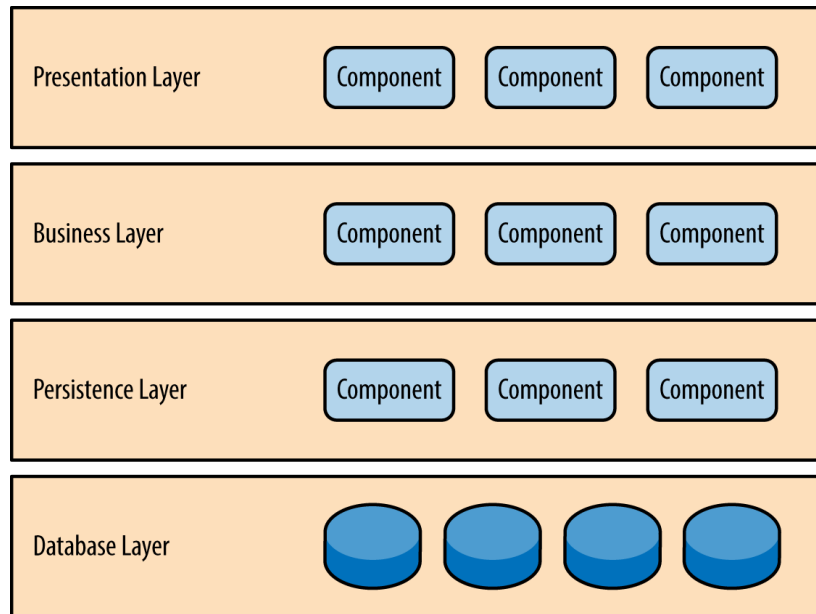
- = Software architecture pattern

(<https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html>)

- Vrijwel standaard in de meeste Java toepassingen
Bijgevolg gekend door veel ontwikkelaars.
- Layered architecture wordt vermeld in de project analyse blueprint.
- Java Spring dwingt eveneens de layered architecture af.

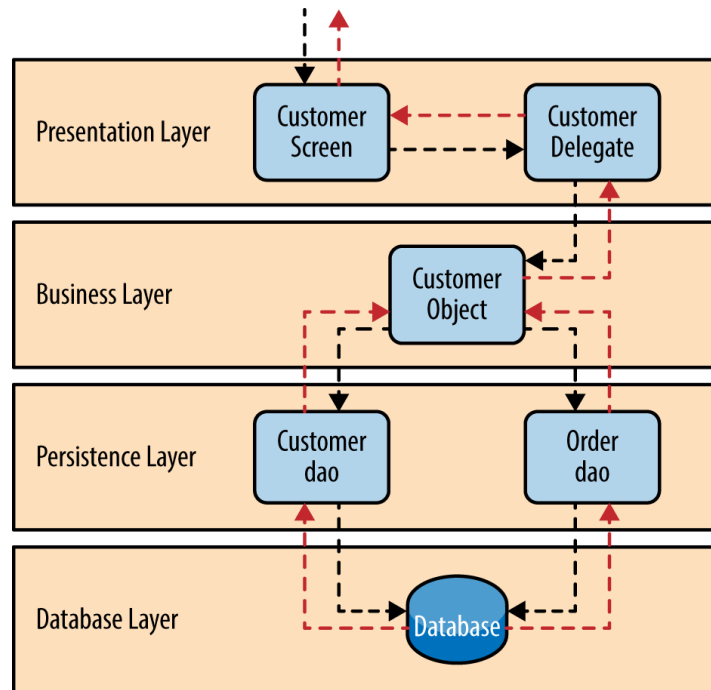
Layers / laagjes

- Toepassing bestaat uit meerdere lagen
- Elke laag heeft een doel
- Elke laag praat enkel met laag direct boven of onder
- Grote toepassingen kunnen meer lagen hebben



Voorbeeld

- Communicatie tussen lagen
- Zwarte pijlen : pad gevolgd om data op te vragen
- Rode pijlen : antwoord dat terug vloeit

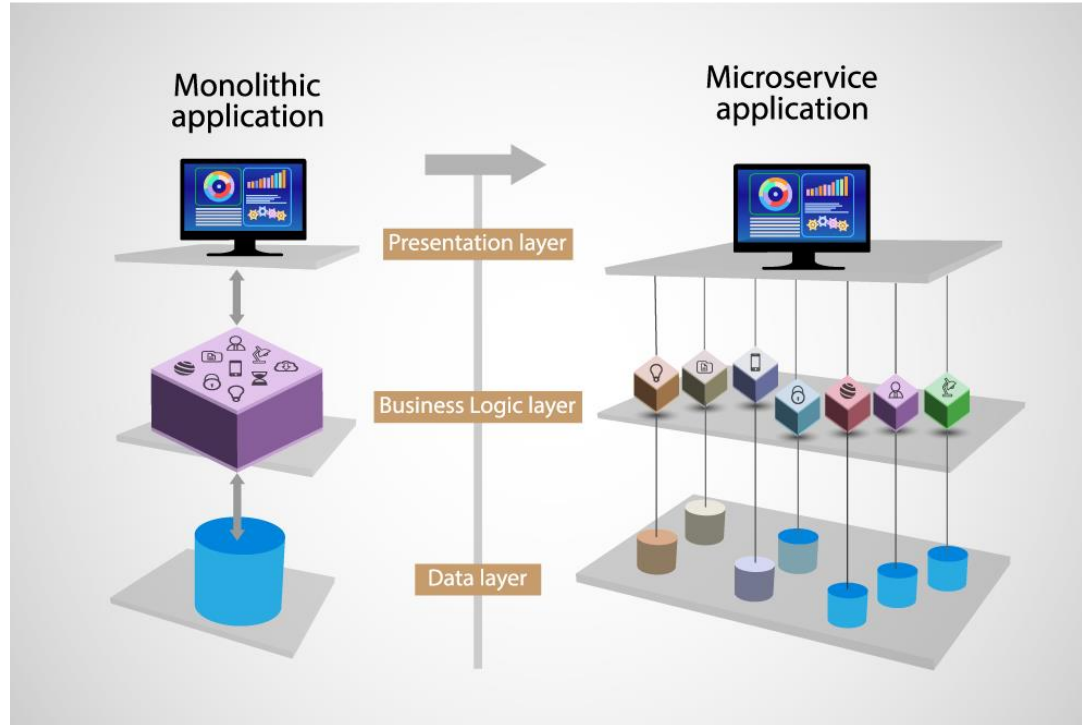


Voordelen

- “Separation of concern” : het is duidelijk wat elke laag moet doen en vooral wat het niet moet doen, waardoor de complexiteit van elk onderdeel laag blijft.
- Testbaarheid : elke laag kan makkelijk getest worden.
- Makkelijk te gebruiken : niet complex en de meeste ontwikkelaars kennen het.

Bedenkingen

1. Layered architecture => monolithic application



Bedenkingen

2. Sinkhole anti-pattern

Als elke laag gewoon data doorgeeft zonder extra handeling.
Het patroon wordt onnuttig en zorgt voor extra overhead.

3. Niet heel agile (monolith)

Bij kleine aanpassingen moet steeds de volledige server gedeployed worden.

Kleine “wijziging” kan veel werk zijn als er in meerdere lagen aanpassingen moeten gebeuren : bv extra kolom in database.

Hibernate herhaling



Hibernate – Stappenplan

1. Maven dependencies toevoegen
 - hibernate
 - database
2. Hibernate database configuratie schrijven
3. Java Object mapping schrijven
4. SessionFactory schrijven
5. Data Access Object (DAO) schrijven

Hibernate database configuration

- Eenmalig aan te maken voor je database
- Twee mogelijkheden voor de configuratie:
 - Configuratie in XML bestand : hibernate.cfg.xml
 - Configuratie in Java code
- **Voorkeur voor XML bestand**
- XML bestand komt in de /resources folder

H2 XML database configuration

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="hibernate.connection.driver_class">org.h2.Driver</property>
        <property name="hibernate.connection.url">jdbc:h2:./data/mydatabase</property>
        <property name="hibernate.connection.username">sa</property>
        <property name="hibernate.connection.password"></property>

        <property name="hibernate.show_sql">>true</property>
        <property name="hibernate.hbm2ddl.auto">create</property>
    </session-factory>
</hibernate-configuration>
```

Default username = sa
Default password = ""

Java Object Mapping

- Java Object koppelen aan een database (= Persistent Class)
- Twee mogelijkheden voor de configuratie
 - Configuratie in XML bestand : *classname.hbm.xml*
 - Configuratie in Java code dmv Java Annotation
- Java Object aanpassen naar Persistent Class regels
- **Voorkeur voor Java Annotation**

Persistent Class regels

1. De class moet een default constructor hebben (en mag daarnaast nog extra constructors hebben).
2. De class moet een ID attribuut hebben.
3. De class attributen moeten private zijn.
4. De class attributen mogen niet final zijn.
5. De class attributen moeten een getter en setter hebben.

Persistent Class regels toepassen

```
public class StormTrooper {  
    private final String name;  
    private final Rank rank;  
  
    public StormTrooper(String name, Rank rank) {  
        this.name = name;  
        this.rank = rank;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public Rank getRank() {  
        return rank;  
    }  
}
```



```
public class StormTrooper {  
    private int id;  
    private String name;  
    private Rank rank;  
  
    public StormTrooper() {  
    }  
  
    public StormTrooper(String name, Rank rank) {  
        this.name = name;  
        this.rank = rank;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
}
```

Object code mapping

```
@Entity
@Table(name = "STORMTROOPER")
public class StormTrooper implements Comparable<StormTrooper> {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;
    @Column(name = "name")
    private String name;
    @Enumerated(EnumType.STRING)
    @Column(name = "rank")
    private Rank rank;

    public StormTrooper() {
    }

    public StormTrooper(String name, Rank rank) {
        this.name = name;
        this.rank = rank;
    }
}
```

Object code mapping

OPGELET : gebruik jakarta.persistence package voor imports !!!

- @Entity : marker annotation om aan te geven dat deze class op een tabel moet gemapped worden
- @Table : tabelnaam
- @Id : marker annotation om aan te geven dat het veld een ID is
- @GeneratedValue : ID generatie strategy
- @Column : kolomnaam
- @Enumerated : mapping van Enum naar kolom datatype

SessionFactory

Communicatie met database = Session

Eénmalig moet je een SessionFactory aanmaken. Dit object is verantwoordelijk voor het toekennen van Session objecten.

HibernateUtil

```
public class HibernateUtil {  
    private static SessionFactory sessionFactory;  
  
    public static SessionFactory getSessionFactory() {  
        if (sessionFactory == null) {  
            try {  
                Configuration configuration = new Configuration();  
                configuration.configure("hibernate.cfg.xml");  
                configuration.addAnnotatedClass(StormTrooper.class);  
                configuration.addAnnotatedClass(Squad.class);  
                sessionFactory = configuration.buildSessionFactory();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
        return sessionFactory;  
    }  
}
```

static SessionFactory

→ eenmalig aangemaakt en beschikbaar doorheen de toepassing

static sessionFactory

→ indien SessionFactory niet bestaat, aanmaken via config

addAnnotatedClass()

→ elke persistent class doorgeven aan de hibernate configuratie

Data Access Object

Java Object dat de communicatie met de database voor één Entity op zich neemt.

Afscheren van database logica (queries)

Naamgeving class : Entity naam + DAO

DAO – SELECT list

```
public class SquadDAO {  
    public List<Squad> getSquads() {  
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {  
            String hql = "SELECT s FROM Squad s";  
            Query<Squad> query = session.createQuery(hql, Squad.class);  
            return query.list();  
        }  
    }  
}
```

List bevat 0 tot n elementen en is nooit null.

Gebruik HQL (Hibernate Query Language) om queries te schrijven.

DAO – find by id with Optional

```
public class SquadDAO {  
    public Optional<Squad> getSquadById(int id) {  
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {  
            return Optional.ofNullable(session.get(Squad.class, id));  
        }  
    }  
}
```

Optional aanmaken

Optional.ofNullable → korte notatie voor :

if(squad == null) return Optional.empty() else return Optional.of(squad)

DAO – create

```
public class SquadDAO {  
    public void createSquad(Squad squad) {  
        Transaction transaction = null;  
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {  
            transaction = session.beginTransaction();  
            session.persist(squad);  
            transaction.commit();  
        } catch (Exception e) {  
            if (transaction != null) {  
                transaction.rollback();  
            }  
        }  
    }  
}
```

DAO – update

```
public class SquadDAO {  
    public void updateSquad(Squad squad) {  
        Transaction transaction = null;  
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {  
            transaction = session.beginTransaction();  
            session.merge(squad);  
            transaction.commit();  
        } catch (Exception e) {  
            if (transaction != null) {  
                transaction.rollback();  
            }  
        }  
    }  
}
```

DAO – delete

```
public class SquadDAO {  
    public void deleteSquad(Squad squad) {  
        Transaction transaction = null;  
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {  
            transaction = session.beginTransaction();  
            session.remove(squad);  
            transaction.commit();  
        } catch (Exception e) {  
            if (transaction != null) {  
                transaction.rollback();  
            }  
        }  
    }  
}
```

Advanced Mapping 'extends' – optie 1

@MappedSuperclass

public abstract class StarWarsCharacter

@Entity

@Table(name="DARTHVADER")

public class DarthVader extends StarWarsCharacter

@MappedSuperclass:

De abstract class wordt zelf géén tabel, de subclasses bevatten alle velden van de abstract class + eigen velden.

Advanced Mapping 'extends' – optie 2

@Entity

@Table(name="STARWARS_CHARACTER")

@DiscriminatorColumn(name="character_type")

public abstract class StarWarsCharacter

@Entity

public class DarthVader extends StarWarsCharacter

Single Table:

Er wordt één tabel aangemaakt met alle attributen van zowel de abstract class als **alle subclasses**.

Er wordt daarnaast ook een extra kolom gemaakt : DiscriminatorColumn die de naam van de subclass bevat

Advanced Mapping : bidirectionele relatie

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "squad_id")
private Squad squad;
```

- @ManyToOne : marker om de relatie te activeren
 - FetchType : EAGER of LAZY
- @JoinColumn : naam van de Foreign Key kolom in de tabel

→ Zelfde notatie voor @OneToOne

Advanced Mapping : bidirectionele relatie

```
@OneToMany(fetch = FetchType.EAGER)
@JoinTable(name = "SQUAD_TROOPER")
private Set<StormTrooper> troopers;
```

- @OneToMany : marker om de relatie te activeren
 - FetchType : EAGER of LAZY
- @JoinTable : naam van de koppeltabel

→ Zelfde notatie voor @ManyToMany



**FOR TODAY
ANYWAY...**