# OBJECT-RELATIONAL MAPPING (ORM)

# ENTITY CLASSES

- POJO
  - hergebruik buiten de context van JPA
  - voorwaarden
    - no-args constructor
    - accessor methods (getters/setters)
    - primary key
    - geen final class/velden
- configureren als entity: @Entity (+@Id)

# ENTITY CLASSES

```
@Entity
public class Message {
    private long id;
    private String text;

    public Message() {
    }

    public Message(long id, String text) {
        this.id = id;
        this.text = text;
    }

    @Id
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

# PRIMARY KEYS

- verplicht voor elke entity class
- garanderen uniqueness in de database
- type
  - primitive (byte, int, long, ..) of wrapper(Integer, Long,..)
  - String
  - maar ook BigInteger, Date
  - of.. samengesteld object

# PRIMARY KEYS

- soorten
  - natuurlijke primary key: betekenisvolle waarde die deel uitmaakt van het object bvb. String rijksregisternummer
  - surrogaat primary key: extra veld zonder andere betekenis bvb. long id, String UUID
- voorwaarden:
  - unieke voorstelling van het object/record
  - waarde wijzigt nooit

# ENKELVOUDIGE PRIMARY KEY

- aanduiden met @Id
- automatisch gegenereerde surrogaat PK
- @GeneratedValue
- eventueel met strategie

```
@Id
@GeneratedValue
public long getId() {
    return id;
}
```

```
@GeneratedValue(strategy=GenerationType.AUTO)
```

# ENKELVOUDIGE PRIMARY KEY

- sommigen enkel mogelijk op bepaalde DBMS
- onmiddellijk vs at commit time

| *Strategie* | *Omschrijving* |
|---|---|
| `IDENTITY` | Er wordt gebruik gemaakt van een *identity column* in de databank. Dit is een kolom waarvan de inhoud automatisch ophoogt bij de creatie van een record (*auto-increment*). |
| `SEQUENCE` | Er wordt gebruik gemaakt van een *sequence* in de databank. |
| `TABLE` | Er wordt gebruik gemaakt van een aparte tabel waarin telkens de hoogste waarde van de *primary key* bewaard wordt. |
| `AUTO` | De JPA-*provider* kiest zelf de beste strategie, rekening houdend met de mogelijkheden van de onderliggende databank. Dit is tevens de standaard waarde. |

```
@SequenceGenerator(name="mySequence", sequenceName="SEQ01")
@GeneratedValue(generator="mySequence")
```

# SAMENGESTELDE PRIMARY KEY

- primary key class definiëren
  - serializable
  - public no-args constructor
    - accessor methods (getters/setters) voor élk element van de PK
    - correcte equals/hashcode implementatie hebben
- (!) de entity class zelf moet exact dezelfde velden hebben die allemaal gemarkeerd zijn met @Id
- de entity class duidt de PK aan met @IdClass

# ORM - PERSISTENTE OBJECTEN

```
@Entity
@IdClass(PlayerPK.class)
public class Player {
   private String club;
   private int number;
   private String name;

   @Id
   public String getClub() {
      return club;
   }
   public void setClub(String club) {
      this.club = club;
   }

   @Id
   public int getNumber() {
      return number;
   }
   public void setNumber(int number) {
      this.number = number;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

# ORM - PERSISTENTE OBJECTEN

```
public class PlayerPK implements Serializable {
   private String club;
   private int number;

   public PlayerPK() {
   }

   public PlayerPK(String club, int number) {
      this.club = club;
      this.number = number;
   }

   public String getClub() {
      return club;
   }
   public void setClub(String club) {
      this.club = club;
   }
   public int getNumber() {
      return number;
   }
   public void setNumber(int number) {
      this.number = number;
   }

   @Override
   public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + ((club == null) ? 0 :
club.hashCode());
      result = prime * result + number;
      return result;
   }

   @Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      PlayerPK other = (PlayerPK) obj;
      if (club == null) {
         if (other.club != null)
            return false;
      } else if (!club.equals(other.club))
         return false;
      if (number != other.number)
         return false;
      return true;
   }
}
```

# ORM - PERSISTENTE OBJECTEN

```
public class SavePlayer {
   public static void main(String[] args) {
      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("course");
      EntityManager em = emf.createEntityManager();
      EntityTransaction tx = em.getTransaction();
      tx.begin();
      Player player = new Player();
      player.setNumber(1);
      player.setClub("FC De Kampioenen");
      player.setName("Markske");
      em.persist(player);
      tx.commit();
      em.close();
      emf.close();
      System.out.println("Player saved.");
   }
}
```

```
public class GetPlayer {
   public static void main(String[] args) {
      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("course");
      EntityManager em = emf.createEntityManager();
      EntityTransaction tx = em.getTransaction();
      tx.begin();
      PlayerPK pk = new PlayerPK();
      pk.setClub("FC De Kampioenen");
      pk.setNumber(1);
      Player player = em.find(Player.class,pk);
      System.out.println(player.getName());
      tx.commit();
      em.close();
      emf.close();
   }
}
```

# OBJECT IDENTITY / EQUALITY

- object identity: zelfde geheugenplaats in de VM : ==
- object equality: op basis van inhoud : equals() + hashCode()
  - degelijke implementatie voorzien! (want standaard voor class Object is object identity)
  - probeer business key te identificeren (essentiële velden die nooit wijzigen)

# OBJECT IDENTITY / EQUALITY

- wat gebeurt er als de business key toch wijzigt…
  - tijdens itereren over collection
- of als de object-graph geen soft-links bevat maar ieder object wel via-via een relatie heeft naar een ander object…
- OPGEPAST: als je enkel je ‘database identity’ opneemt kan je ongewenste side-effects krijgen als je object nog niet gepersisteerd geweest is
  - UUID’s to the rescue?
- (!) contract: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

# FIELD/PROPERTY ACCESS

- FIELD ACCESS:
  - variabelen rechtstreeks gemanipuleerd door JPA
  - getters/setters genegeerd
  - voordeel: bepaalde fields niet exposed bvb. geen setId() method aka readable/writable properties
- PROPERTY ACCESS:
  - JPA gebruikt de accessor methods (getters/setters)
  - voordeel (?): eventuele extra code in getter/setter wordt uitgevoerd
- wordt bepaald door plaats @Id
  - op het veld: FIELD
  - op de setter: PROPERTY

# VOORBEELD FIELD ACCESS

```
@Entity
public class Message {
    @Id
    private long id;
    private String text;

    public Message() {
    }

    public Message(long id, String text) {
        this.id = id;
        this.textveld = text;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getText() {
        return textveld;
```

# ENTITY MANAGER

- regelt de communicatie van de POJO entities met de database
- synchroniseert de gegevens van een object met de databank
- beheert verzameling van objecten uit 1zelfde persistence unit
- bekomen via EntityManagerFactory (implementatie afhankelijk va de persistence provider)

```
EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("course");
EntityManager em = emf.createEntityManager();
```

# ENTITY MANAGER

PERSISTENCE UNIT (PERSISTENCE.XML)

PERSISTENCE

ENTITYMANAGER FACTORY

ENTITYMANAGER

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
   version="2.1">
   <persistence-unit name="course"
                     transaction-type="RESOURCE_LOCAL">
      <properties>
         <property name="javax.persistence.jdbc.driver"
              value="com.mysql.jdbc.Driver" />
         <property name="javax.persistence.jdbc.url"
              value="jdbc:mysql://noelvaes.eu:3306/StudentDB" />
         <property name="javax.persistence.jdbc.user"
              value="student" />
         <property name="javax.persistence.jdbc.password"
              value="student123" />
         <property
         name="javax.persistence.schema-generation.database.action"
```

# PERSISTENCE CONTEXT

- verzameling objecten die op een bepaald moment onder de hoede van een entity manager vallen
- objecten kunnen toegevoegd worden
  - persist()
  - find()
- objecten die zich in de PC vinden, noemen we MANAGED
- objecten buiten de PC noemen we UNMANAGED
- MANAGED wordt UNMANAGED van zodra de PC afgesloten wordt (close(), detach() of clear())

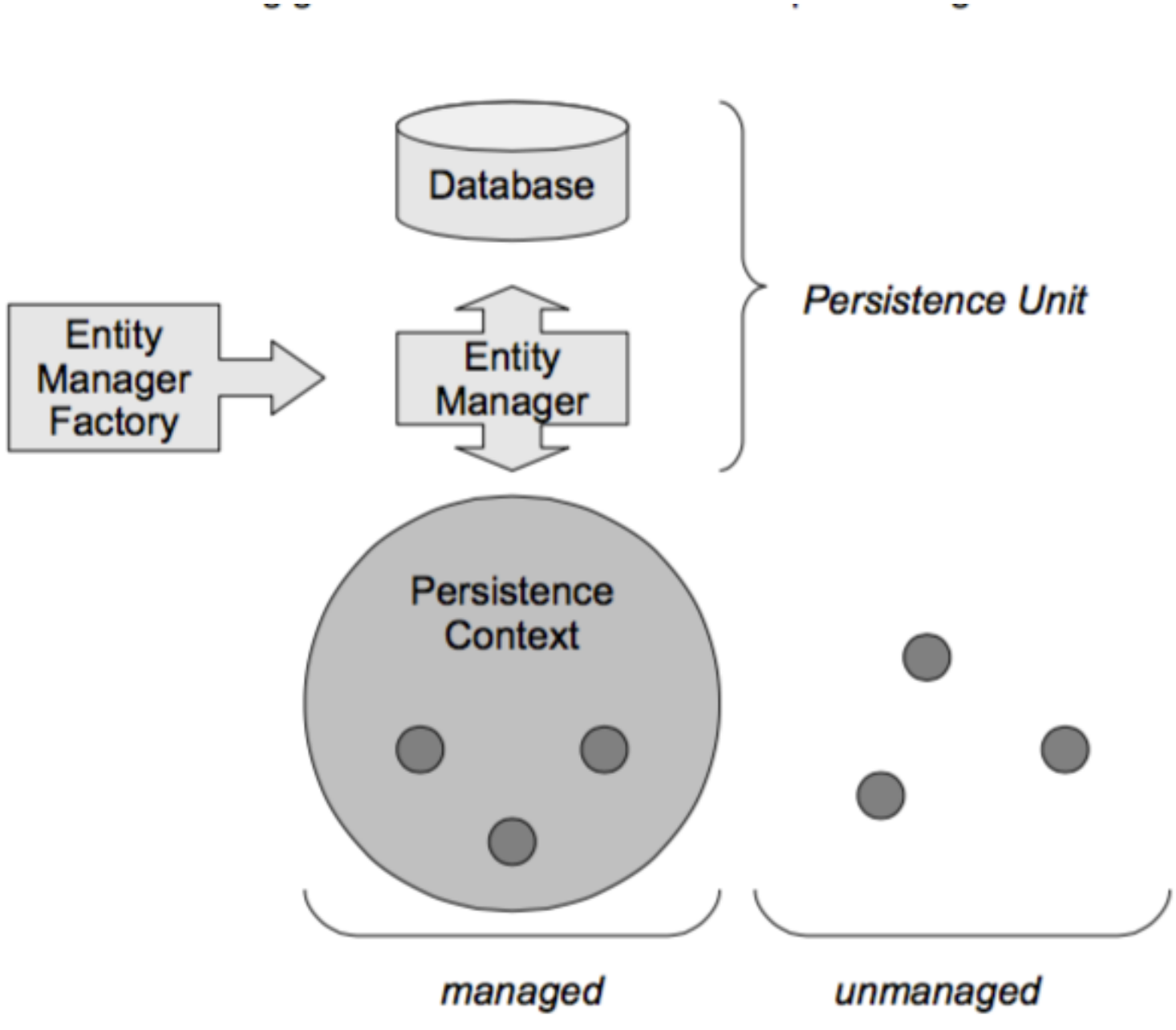
# PERSISTENCE CONTEXT

- wijzigingen die gebeuren op managed objects
  - worden naar databank weggeschreven
    - moment afhankelijk van transactie-beheer
- automatisch dirty checking door entity manager

```
public class ChangeMessage {
   public static void main(String[] args) {
      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("course");
      EntityManager em = emf.createEntityManager();
      EntityTransaction tx = em.getTransaction();
      tx.begin();
      Message  message = em.find(Message.class,1L);
      message.setText("Hello Mars");  // managed
      tx.commit();
      em.close();
      message.setText("Hello Venus"); // unmanaged
      emf.close();
   }
}
```

```
tx.begin();
Message  message = em.find(Message.class,1L);
em.detach(message);
message.setText("Hello Mars"); // unmanaged !!
```

# PERSISTENCE CONTEXT



# TRANSACTIES

- gedrag van objecten binnen PC hangt af van het transactie-beheer
- ACID transacties
  - unit of work
  - Atomair: all or nothing principe
  - Consistent: onderliggende data storage
  - Isolated: niet beïnvloed door andere trx
  - Durable: veilig bewaard, overleeft systeem crash

# JPA TRANSACTIES

- transactie moet expliciet gestart en voltooid worden
- via EntityManager
  - automatische dirty checking
- wijzigingen binnen een persistence context kunnen enkel gebeuren via een transactie
  - dan worden de “managed objects” naar database geschreven (commit())
    - of ook tijdens flush()
    - boundary tussen managed / detached object

# JPA TRANSACTIES

```
EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("course");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Message  message1 = em.find(Message.class,1L);
System.out.println(message1.getText());
Message  message2 = em.find(Message.class,2L);
System.out.println(message2.getText());
tx.commit();
em.close();
emf.close();
```

# JPA TRANSACTIES

- vb. : managed, toch niet meer naar database gecommit

```
EntityManagerFactory emf = Persistence
      .createEntityManagerFactory("course");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Message message = em.find(Message.class,1L);
message.setText("Hello Mars");
tx.commit();
message.setText("Hello Venus");   // managed !!
em.close();
emf.close();
```

# JPA TRANSACTIES

- vb. : managed, uiteindelijk via 2e transactie alsnog naar database

```
EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("course");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Message  message = em.find(Message.class,1L);
message.setText("Hello Mars");
tx.commit();
message.setText("Hello Venus");   // managed !!
tx.begin();
tx.commit();
em.close();
emf.close();
```

# JPA TRANSACTIES

- rollback kan natuurlijk ook ...

```
EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("course");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Message  message = em.find(Message.class,1L);
message.setText("Hello Mars");
tx.rollback();
em.close();
emf.close();
```

# ENTITY MANAGER

- bewaren : persist()
- zoeken: find()
- (!) samenvoegen managed/unmanaged: merge()
- verwijderen: remove()

Indien een object terug *unmanaged* wordt, zullen de wijzigingen die nadien gebeuren niet naar de databank weggeschreven worden. Het is evenwel wel mogelijk de gegevens van een *unmanaged* object samen te voegen met de gegevens van een *managed* object. In feite worden hierbij de gegevens van het *unmanaged* object gewoon gekopieerd in het *managed* object.

```
<T> T find(Class<T> entityClass, Object primaryKey)
<T> T getReference(Class<T> entityClass, Object primaryKey)

<T> T merge(T entity)
```

# ENTITY MANAGER

Objecten kunnen op verschillende manieren verwijderd worden uit de *peristence context*. We sommen even de mogelijkheden op:

1. Bij het afsluiten van de *entity manager* met de methode `close()`. Alle objecten worden dan automatisch terug *unmanaged*.
2. Bij het terugdraaien van een transactie (*rollback*). Ook hier worden alle objecten automatisch verwijderd uit de *persistence context*.
3. Bij het expliciet leegmaken van de volledige *persistence context* met de methode `clear()`. Alle objecten worden dan *unmanaged*.
4. Bij het verwijderen van een object uit de *persistence context* met de methode `detach()`. Alleen het meegegeven object wordt dan *unmanaged*.
5. Bij het verwijderen van een object met de methode `remove()`. De gegevens worden verwijderd uit de databank en het overeenkomstige object wordt *unmanaged*.

# DOMEIN MODEL

- geheel van objecten en hun onderlinge relaties
- mapping
  - van uit het domein model: JPA creëert tabellen
  - van uit het relationele model: bovenop een reeds bestaande database

```
...
<property
 name="javax.persistence.schema-generation.database.action"
 value="none" />
...
```

We geven nog even een overzicht van de mogelijke instellingen:

| ***Waarde*** | ***Betekenis*** |
|---|---|
| `none` | Er wordt geen actie ondernomen. |
| `create` | De tabellen worden automatisch aangemaakt. |
| `drop-and-create` | De tabellen worden verwijderd en opnieuw aangemaakt. |
| `drop` | De tabellen worden verwijderd. |

# @TABLE

```
@Entity
@Table(name="PERSONS")
public class Person {
    ...
}
```

```
@Entity
@Table(name = "PERSONS" ,
        indexes={@Index(name="LAST_NAME_INDEX",
                        columnList="lastName")})
public class Person {
    ...
}
```

## @COLUMN

```
@Id
@GeneratedValue
private long id;
@Column(name="FIRST_NAME")
private String firstName;
@Column(name="LAST_NAME")
private String lastName;
```

| *Element* | *Omschrijving* |
|---|---|
| `columnDefinition` | SQL-fragment dat gebruikt wordt bij de DDL voor de kolom. |
| `insertable` | Geeft aan of dit veld wordt opgegeven bij het invoegen van een nieuw record. Standaard is `true`. |
| `length` | De lengte van de kolom. |
| `name` | De naam van de kolom. |
| `nullable` | Geeft aan of de kolom `NULL`-waarden mag bevatten. Standaard is `true`. |
| `precision` | De nauwkeurigheid van getallen in geval van een decimale waarde. |
| `scale` | De schaal van getallen in geval van een decimale waarde. |
| `table` | De naam van de (secundaire) tabel die dit veld bevat. |
| `unique` | Geeft aan of dit veld uniek moet zijn. Standaard `false`. |
| `updatable` | Geeft aan of dit veld aangepast moet worden bij een *update*. Standaard `true`. |

Enkelvoudige datatype kunnen voorzien worden van de annotatie `@Basic` waarmee wordt aangegeven dat het een basisveld is.

Doorgaans wordt deze annotatie weggelaten, tenzij men gebruik wil maken van een van volgende elementen:

| ***Annotatie*** | ***Omschrijving*** |
|---|---|
| `@Basic(optional)` | Hiermee geven we aan of deze waarde optioneel is; m.a.w. of een lege waarde is toegestaan. Mogelijke waarden zijn `true` en `false`. De standaard waarde is `true`.<br>Voorbeeld: `@Basic(optional=false)` |
| `@Basic(fetch)` | Hiermee geven we aan of deze waarde onmiddellijk of vertraagd gelezen dient te worden. Bij vertraagd lezen, zullen de gegevens maar uit de databank gelezen worden zodra ze voor de eerste keer worden opgevraagd d.m.v. een *getter*. Mogelijke waarden zijn `FetchType.LAZY` en `FetchType.EAGER`. De standaard waarde is `EAGER`. Merk op dat dit slechts een aanbeveling en geen onvoorwaardelijke verplichting is voor de implementatie van het persistentiemechanisme. Het *fetch*-type heeft de bedoeling performantie-optimalisaties mogelijk te maken.<br>Voorbeeld: `@Basic(fetch=FetchType.LAZY)` |

#### 3.3.3.2.2 Datums en tijden

Voor datums en tijden kunnen volgende types gebruikt worden:

| *Java type* |
|---|
| `java.util.Date` |
| `java.util.Calendar` |
| `java.sql.TimeStamp` |

Bij het gebruik van de Java datatypes uit het pakket `java.util` kan men zowel een datum, een tijd of beide bedoelen. Om dit aan te geven dient men de annotatie `@Temporal` aan het persistent veld toe te voegen. Voor de versies uit het pakket `java.sql` is dat niet nodig omdat hier het datatype zelf aangeeft of het om een datum, tijd of datum met tijd gaat.

| *Annotatie* | *Omschrijving* |
|---|---|
| `@Temporal(TemporalType.DATE)` | Enkel de datum. |
| `@Temporal(TemporalType.TIME)` | Enkel de tijd. |
| `@Temporal(TemporalType.TIMESTAMP)` | Datum en tijd. |

## ORM - MAPPING VAN DATATYPES

| ***Annotatie*** | ***Omschrijving*** |
|---|---|
| `@Enumerated(EnumType.ORDINAL)` | Het volgnummer. Standaard waarde. |
| `@Enumerated(EnumType.STRING)` | De *string*-representatie |

```
package persons;

public enum GenderType {
    MALE,FEMALE
}
```

Vervolgens voegen we een persistent veld toe in de klasse ***Person***:

```
    @Enumerated(EnumType.STRING)
    private GenderType gender;
```

In de databank zal deze waarde nu opgeslagen worden als "`MALE`" of "`FEMALE`".

### 5.6.3.2.6. Velden uitsluiten en virtuele velden

Om een veld uit te sluiten als persistent veld kunnen we de annotatie `@Transient` gebruiken. Deze kan geplaatst worden voor een variabele of een *getter*.

```
...
@Transient
public int getAge() {
   Calendar today = new GregorianCalendar();
   return today.get(Calendar.YEAR)-birthDay.get(Calendar.YEAR);
}
...
```