



# Hibernate JPA Schnelleinstieg

**Michael Inden**  
**Freiberuflicher Consultant und Trainer**

---

# Speaker Intro



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre SSE bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre TPL, SA bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre LSA / Trainer bei Zühlke Engineering AG in Zürich
- ~3 Jahre TL / CTO bei Direct Mail Informatics / ASMIQ in Zürich
- Freiberuflicher Consultant, Trainer und Konferenz-Speaker
- Autor und Gutachter beim dpunkt.verlag

E-Mail: [michael.inden@hotmail.ch](mailto:michael.inden@hotmail.ch)  
Blog: <https://jaxenter.de/author/minden>

Kurse: Bitte sprecht mich an!

<https://github.com/Michaeli71/JPA-Workshop.git>





# Agenda



- **PART 1: Einstieg Persistenz früher und heute sowie Herausforderungen**
  - Überblick, Einsatzgebiete
- **PART 2: Basisbausteine & Schnelleinstieg**
  - EntityManagerFactory, EntityManager, Persistence Unit, Entity, Entity Lifecycle
  - JPA im Kontext von JavaSE / Java EE
  - Definition von Persistence Units
  - Generieren des Datenbankschemas für Prototyping
- **PART 3: Einstieg ORM (Object-Relational Mapping)**
  - Annotations & einfache Objekte mit JPA mappen
  - Generierte und zusammengesetzte Schlüssel
  - ID-Generatoren

# Workshop Contents

---



- **PART 4: More ORM**
  - Beziehungen: 1:1, 1:n , n:m
  - Mapping von Objektstrukturen, Listen und Mengen
  - Abbildung von Vererbung
  - **Eager / Lazy Loading, Cascading**
  - **EntityGraph**
- **PART 5: JPQL – Query Language**
  - Grundlagen
  - Unterschiede zu SQL
  - Parametrisierte Queries
  - JOIN FETCH

# Workshop Contents

---



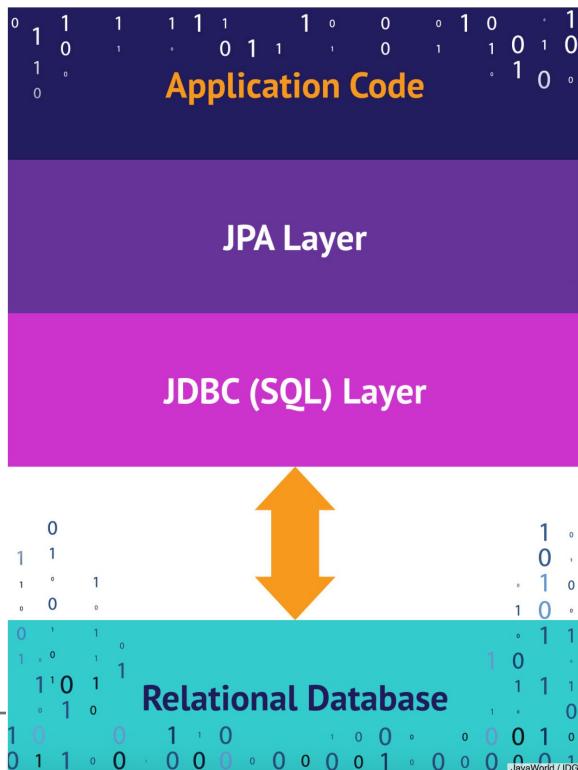
- **PART 6: Transaktionen**
    - Einsatz von JTA, Manuelles Transaktionshandling
    - Pessimistic / Optimistic Locking
  - **PART 7: DAOs und Repositories**
    - Konzept der deklarativen Repositories
  - **PART 8: EntityListener & validation**
    - Add-On: **Entity Listener: Callback-Methoden**
    - Add-On: **Validation**
  - **PART 9: Caching**
    - Add-On: **Caching: First & Second Level Cache**
-



# PART 1: Einführung

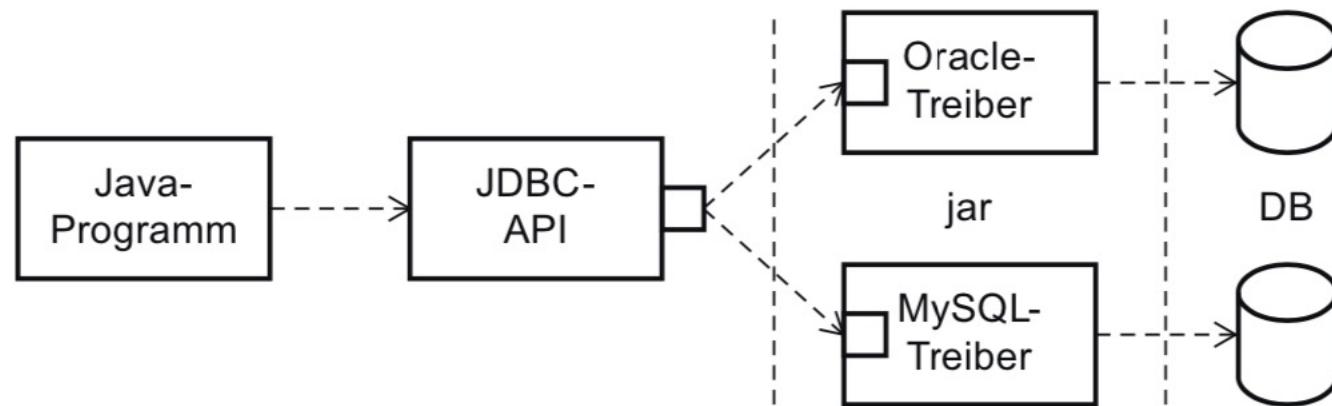


- Java Persistence API (JPA) ist eine Sammlung von Klassen und Interfaces zur Verwaltung von Daten in einer Datenbank
- JPA "vermittelt" zwischen Application / Domain Model und Datenbanken (RDBMS)





- **JDBC = Java Database Connectivity**
- **Definiert einen Standard zum Zugriff auf relationale Datenbanken**

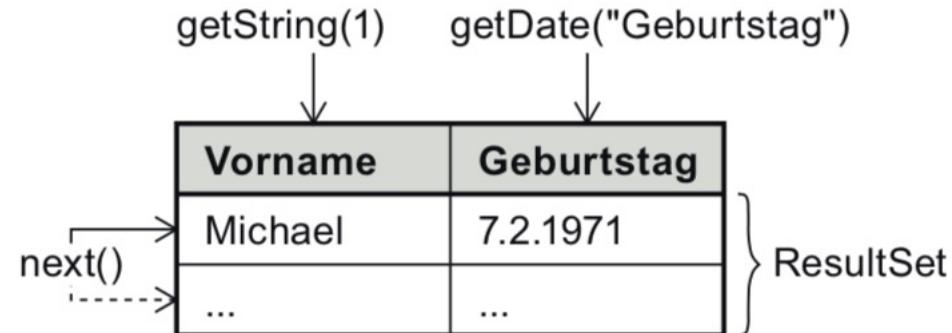


- **javax.sql.DataSource repräsentiert eine abstrakte Fabrik für Verbindungen zu Datenbanken**



- **java.sql.Connection** repräsentiert eine Verbindung zu Datenbank und erzeugt u.a. Statements und steuert Transaktionen usw.
- **java.sql.Statement / PreparedStatement** repräsentiert eine Anweisung, die ausgeführt werden soll
- **java.sql.ResultSet** repräsentiert Ergebnisse von Abfragen

SELECT Vorname, Geburtstag FROM Personen;





- **JPA deutlich mehr high-level als JDBC**
  - Keine JDBC/SQL-Anweisungen durch Entwickler mehr nötig
  - Viele JDBC/SQL-Kommandos werden automatisch durch JPA generiert
  - JPA ist datenbankunabhängig, passt das SQL dann datenbankspezifisch an
  - Für Abfragen SQL ähnliche Sprache (JPQL)
  - Datenbank scheint zwar durch, aber viel mehr Abstraktion
- **JPA dient zum Object/Relational Mapping (ORM)**
  - Applikation arbeitet mit Objekten und dem Konzept der Entities
  - Entities werden auf Datenbanktabellen abgebildet
  - Kommandos und Aktionen als Methoden und nicht SQL-Befehle
  - Assoziationen und Vererbung lassen sich ohne Tricks realisieren



- JPA gleicht mögliche Änderungen am Objekt-/Entity-Zustand automatisch mit der Datenbank ab (am Transaktionsende)
  - Benutzer ändert also im Objektmodell und dies wird gleich in der Datenbank abgebildet, keine zusätzlichen SQL-Befehle nötig
  - Selbst anspruchsvolle Dinge wie Assoziationen und Vererbung werden problemlos unterstützt:
    - Automatischen Laden von referenzierten Objekten möglich
    - Automatisches Löschen von referenzierten Objekten möglich
  - JPA verwaltet die Entities und speichert diese zwischen (1st Level Caching)
-



- Historisches

- JPA 1.0: JSR 220 EJB 3.0 Mai 06
  - JPA 2.0: JSR 317 Java Persistence 2.0 (Java EE 6) Dez. 09
  - *JPA 2.1: JSR 338 Java Persistence 2.1 (Java EE 7)* Apr. 13  
=> Vor Java 8, daher Date and Time API nur über Converter
  - **JPA 2.2: JSR 338 Java Persistence 2.2 (Java EE 8)** Aug. 17  
=> Nach Java 8, mit Unterstützung für diverse Typen aus Date and Time API
  - Jakarta Persistence 3.0 Okt. 20



- JPA ist (nur) eine Spezifikation (keine Implementation)
- JPA Funktionalität wird durch verschiedene Provider implementiert
  - Hibernate
  - EclipseLink / TopLink
  - ObjectDB
- Vergleich von DBs finden sich hier (allerdings 2012)
  - <https://www.jpab.org/>
  - <https://www.jpab.org/All/All/All.html>



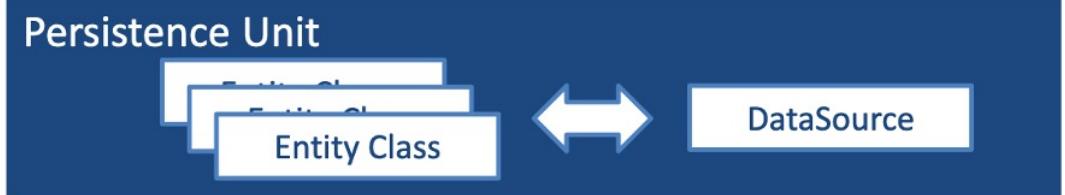
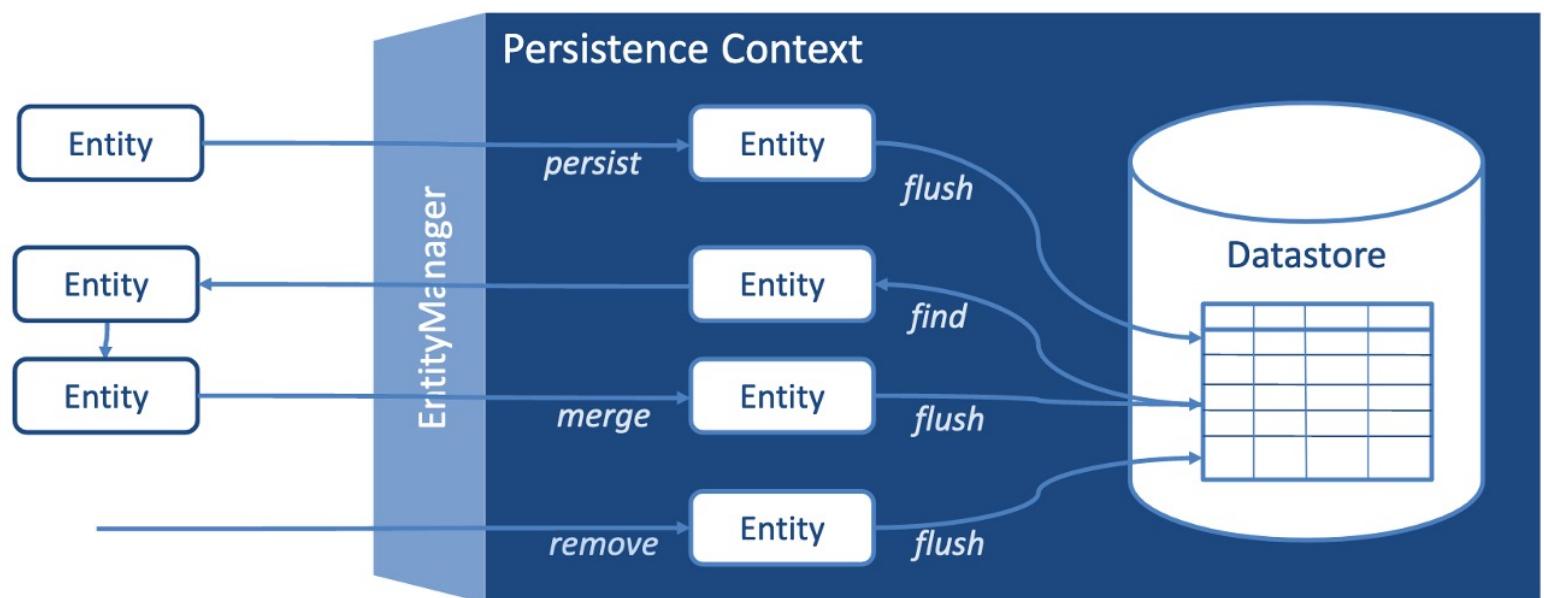
# **PART 2:**

# **Basisbausteine & Schnelleinstieg**

---



- **EntityManager = Verbindung zur Datenbank**
- **Entity = Modell einer Datenbanktabelle**
- **Persistence Context = Menge der von einem EntityManager verwalteten Entities**
- **Persistence Unit = Konfiguration der Verbindung zur DB (persistence.xml)**





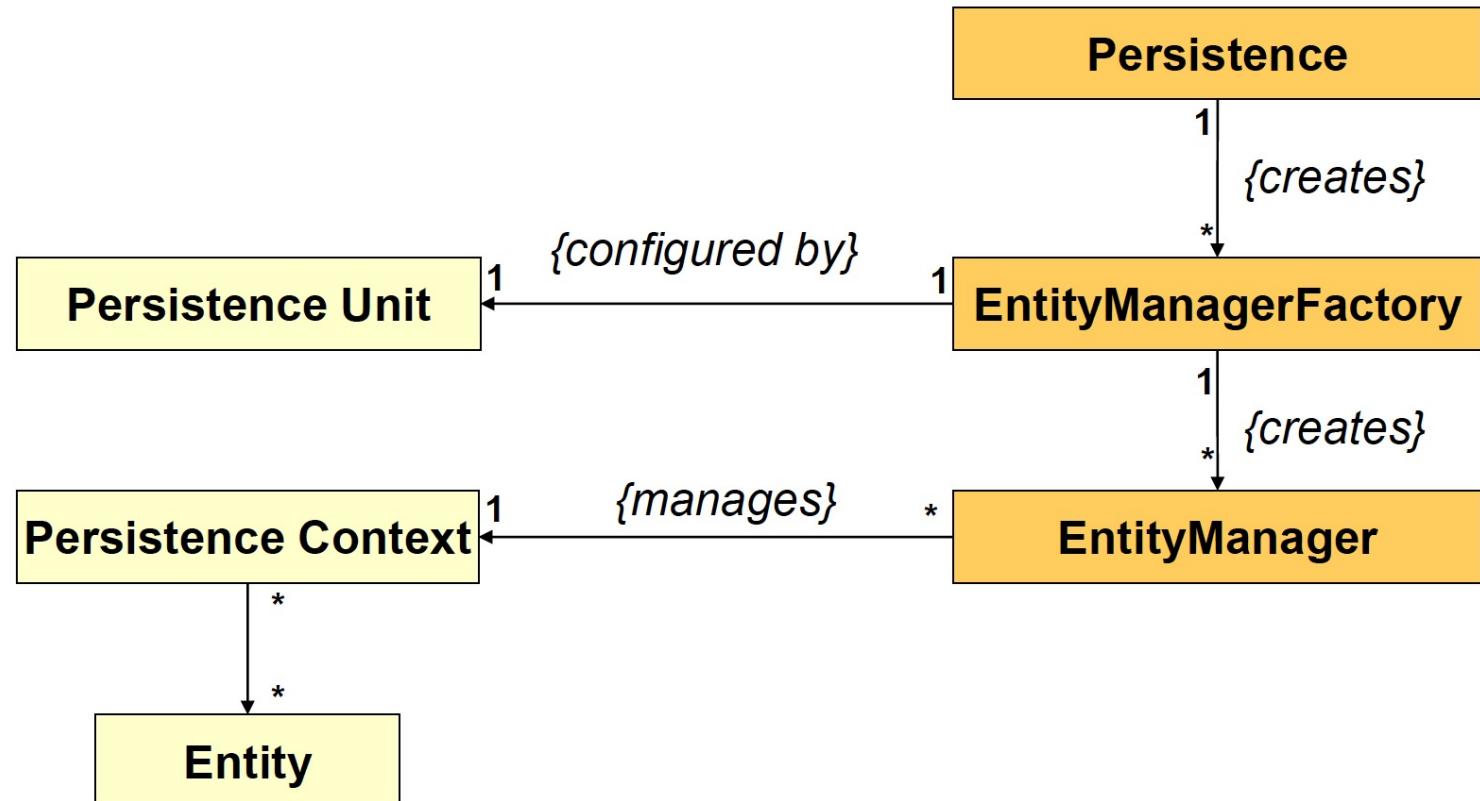
# Woher bekomme ich den EntityManager?



# JPA Hauptkomponenten



- **EntityManagerFactory = Erzeugung von EntityManagers**
- **Und woher erhält man diese?**
  - JavaEE: EntityManagerFactory wird injiziert
  - JavaSE: Persistence = Bootstrap-Klasse für Java SE





# PART 2: Basisbausteine

# Persistence Unit (META-INF/persistence.xml)



```
<persistence-unit name="java-profi-PU-BASICS" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>basics.Account</class>
    <class>basics.AccountSettings</class>
    <class>basics.Movie</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>

    <properties>
        <property name="javax.persistence.jdbc.driver"
                  value="org.hsqldb.jdbcDriver" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        <property name="javax.persistence.jdbc.password" value="" />
        <property name="javax.persistence.jdbc.url"
                  value="jdbc:hsqldb:hsq://localhost/java-profi" />
        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.HSQLDialect" />
    </properties>
</persistence-unit>
```

ACHTUNG: Nur für Tests und Demos solche Werte nutzen!

Oftmals benötigt es hier die Angaben eines Ports.  
Das variiert von DB zu DB



## JAVA SE

```
EntityManagerFactory entityManagerFactory =  
    Persistence.createEntityManagerFactory("java-profi-PU-BASICS");  
  
EntityManager entityManager = entityManagerFactory.createEntityManager();
```

## JAVA EE (injected by container)

```
@PersistenceContext(name="java-profi-PU-BASICS")  
EntityManager entityManager  
  
// Eher ungewöhnlich, da man dann wieder vieles von Hand machen muss  
//@PersistenceUnit(name="java-profi-PU-BASICS")  
//EntityManagerFactory entityManagerFactory
```

- **em und emf als gebräuchliche Abkürzungen**

# Entity Quickstart



- Entity = POJO + @Entity + @Id + weitere Annotations

@Entity

```
public class Movie
{
    @Id
    private Long id;
    private String title;
    private boolean rented;
    private LocalDate releaseDate;

    protected Movie() {}
    public Movie(Long id, String title, boolean rented, LocalDate releaseDate) {
        ..
    }

    public boolean isRented() {
        return rented;
    }

    public void setRented(boolean rented) {
        this.rented = rented;
    }
}
```





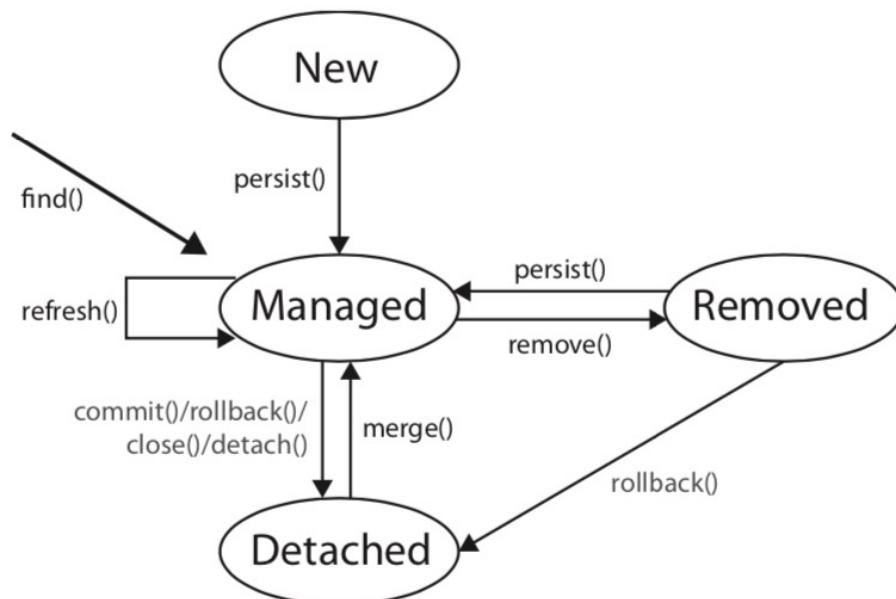
- **Vielzahl an Methoden**
- **CRUD Operationen (persist / find / «update» / remove) & Queries**

```
public interface EntityManager {  
    public void persist(Object entity);  
    public <T> T merge(T entity);  
    public void remove(Object entity);  
    public <T> T find(Class<T> entityClass, Object primaryKey);  
  
    public void flush();  
    public void setFlushMode(FlushModeType flushMode);  
    public FlushModeType getFlushMode();  
  
    public void refresh(Object entity);  
    public void clear();  
    public boolean contains(Object entity);  
    public void detach(Object entity);  
  
    public Query createQuery(String qlString);  
    public Query createNamedQuery(String name);  
    ...
```



- **EntityManager verwaltet Entitäten**

- **NEW** – Eine Instanz einer Entitätsklasse wurde neu erzeugt und ist noch keinem EntityManager zugeordnet, wird folglich auch noch nicht mit der Datenbank abgeglichen.
- **MANAGED** – Eine Instanz einer Entitätsklasse ist Bestandteil eines Persistence Context und wird von einem EntityManager mit der Datenbank abgeglichen.
- **REMOVED** – Eine Entity in diesen Zustand wird später aus der Datenbank entfernt.
- **DETACHED** – Eine Entity im Zustand detached bedeutet, dass diese nicht Bestandteil eines Persistence Context ist und damit nicht mehr vom EntityManager verwaltet wird.



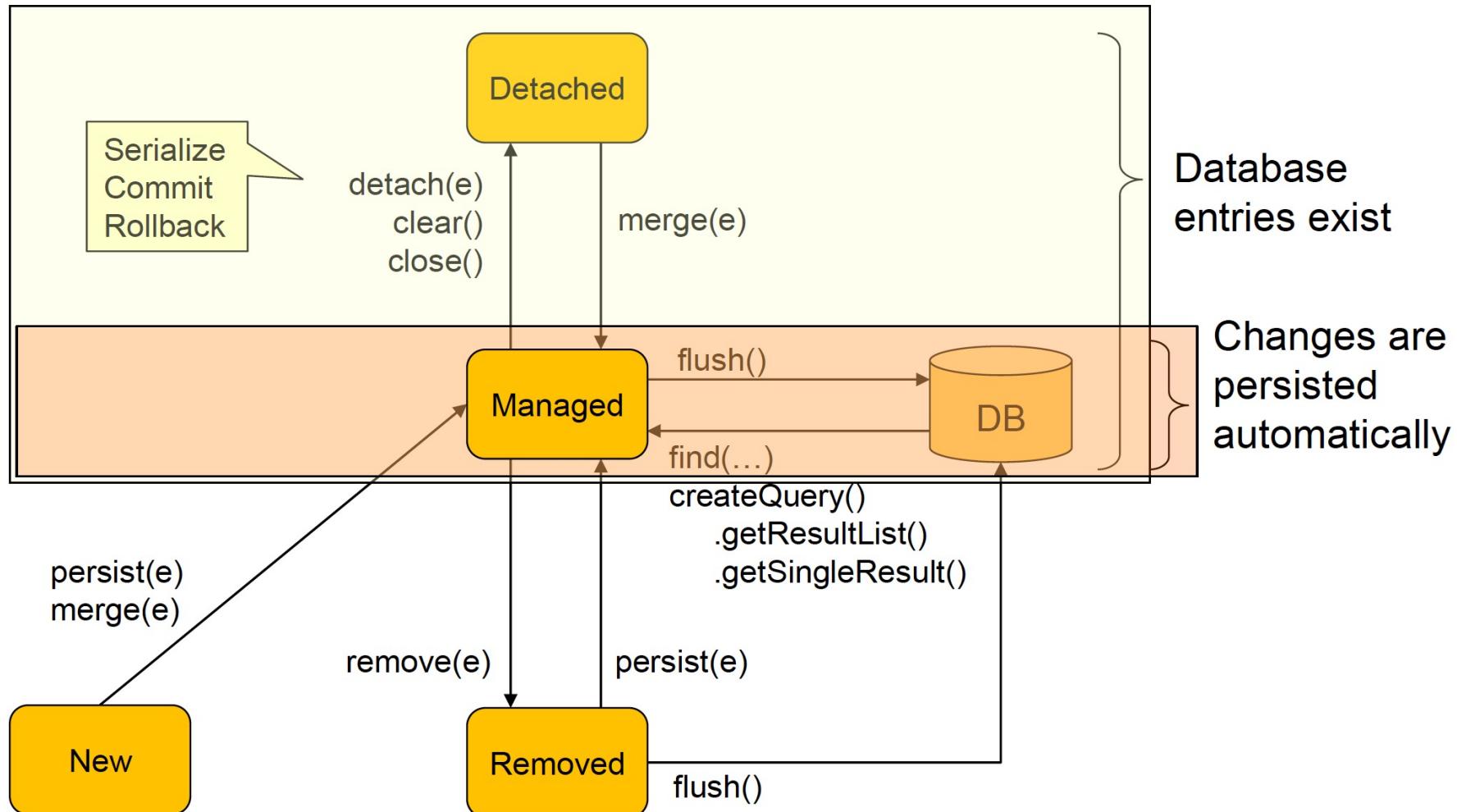
# EntityManager Quickstart



- **`persist(Object entity)`** – Speichert die Entity in der Datenbank und überführt sie in einen vom EntityManager verwalteten Zustand.
- **`find(Class<T> entityClass, Object primaryKey)`** – Sucht basierend auf dem übergebenen Primärschlüssel nach einem Datensatz in der Datenbank und gibt diesen als verwaltete Entity zurück.
- **`detach(Object entity)`** – Löst für die Entity deren Verbindung zum EntityManager.
- **`merge(T entity)`** – Diese Methode dient dazu, vom EntityManager losgelöste Instanzen wieder in den vom EntityManager verwalteten Persistence Context aufzunehmen.
- **`remove(Object entity)`** – Entfernt eine Entity aus dem Persistence Context und später aus der Datenbank (sofern nicht nochmals persist() aufgerufen wird).
- **`refresh(Object entity)`** – Aktualisiert die Entities mit den Werten aus der Datenbank
- **`flush()`** – Speichert die Werte aus den Entities (den Persistence Context) in der Datenbank



# Entity Bean Lifecycle



# EntityManager Query Quickstart



- `createQuery()`
- `createNamedQuery()`
- `createNativeQuery()`
- Parameterübergabe entweder positionsbasiert oder benannt
- positionsbasiert ggf. Gefahr von Verwechslung
- benannt länger, aber verständlicher

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")  
        .setParameter(1, name)  
        .getResultList();  
}  
  
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")  
        .setParameter("custName", name)  
        .setMaxResults(10)  
        .getResultList();  
}  
  
@NamedQuery(  
    name="findAllCustomersWithName",  
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"  
)
```



---

# Persistence Units



# Verweis auf Persistenz-Framework und Klassen



```
<persistence-unit name="java-profi-PU-EXERCISES" transaction-type="RESOURCE_LOCAL">  
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>  
  
    <class>a_exercises.Point</class>  
    ...  
    <class>a_exercises.Project</class>  
    <exclude-unlisted-classes>true</exclude-unlisted-classes>  
  
    <properties>  
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />  
        ...  
    </properties>  
  
</persistence-unit>
```

# Angabe der Datenbankverbindungsdaten



```
<persistence-unit name="java-profi-PU-EXERCISES" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    ...
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        <property name="javax.persistence.jdbc.password" value="" />
        <property name="javax.persistence.jdbc.url"
                  value="jdbc:hsqldb:hsq://localhost/java-profi" />
        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.HSQLDialect" />
    ...
</properties>
</persistence-unit>
```

# Mehrere Persistence Units und sogar mehrere DBs möglich



```
<persistence-unit name="java-profi-PU-EXERCISES" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    ...
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        ...
    </properties>
</persistence-unit>

<persistence-unit name="java-profi-PU-VALIDATION" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    ...
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
        <property name="javax.persistence.jdbc.url"
                  value="jdbc:postgresql://localhost/michaeli" />
        <property name="javax.persistence.jdbc.user" value="michaeli" />
        ...
    </properties>
</persistence-unit>
```

# Protokollierung von DB-Aktionen aktivieren



```
<persistence-unit name="java-profi-PU-EXERCISES" transaction-type="RESOURCE_LOCAL">
    ...
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        ...
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        <property name="javax.persistence.schema-generation.database.action"
                  value="drop-and-create" />
    </properties>
```

Hibernate:

```
select
    point0_.id as id1_0_,
    point0_.x as x2_0_,
    point0_.y as y3_0_
from
    Point point0_
```

# Generieren von DB Schema



```
<persistence-unit name="java-profi-PU-EXERCISES" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    ...
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        ...
        <property name="javax.persistence.schema-generation.database.action"
            value="drop-and-create" />
    </properties>
```

- |                        |   |
|------------------------|---|
| <b>none</b>            | No schema creation or deletion will take place.   |
| <b>create</b>          | The provider will create the database artifacts on application deployment.<br>The artifacts will remain unchanged after application redeployment. |
| <b>drop-and-create</b> | Any artifacts in the database will be deleted, and the provider will create<br>the database artifacts on deployment.                              |
| <b>drop</b>            | Any artifacts in the database will be deleted on application deployment.  |

Hibernate:

```
create table Employee (
    id bigint not null,
    birthday date,
    name varchar(255),
    updated_on timestamp,
    primary key (id)
)
```

# Generieren von DB Schema mit Skripts und Metadaten



```
<persistence-unit name="java-profi-PU-EXERCISES" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    ...
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        ...
        <property name="javax.persistence.schema-generation.database.action"
            value="drop-and-create" />
        <property name="javax.persistence.schema-generation.create-source"
            value="script-then-metadata" />
        <property name="javax.persistence.schema-generation.create-script-source"
            value="META-INF/part2_basics/create.sql" />
    </properties>
```

## Skript: create.sql



```
create table account_settings (
    id bigint not null,
    name varchar(275) not null,
    value varchar(255) not null,
    account_id bigint not null,
    primary key (id)
)

create table accounts (
    id bigint not null,
    email_address varchar(300),
    name varchar(100) not null,
    primary key (id)
)

alter table account_settings
    add constraint FK54uo82jnot7ye32pyc8dcj2eh
    foreign key (account_id)
    references accounts (id)
```



## Skript: create.sql



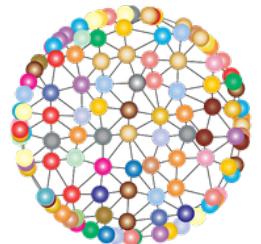
```
create table account_settings (
    id bigint not null,
    name varchar(275) not null,
    value varchar(255) not null,
    account_id bigint not null,
    primary key (id)
)
```



ACHTUNG: MUSS  
EINZEILIG  
ANGEGEBEN WERDEN!



```
CREATE TABLE Persons2 (PersonID int, LastName varchar(255), FirstName varchar(255), ...
CREATE TABLE Movie2 (id bigint not null, releaseDate date, rented boolean not null, ...)
```



**Wie ist das eigentlich mit  
Connection Pooling?**

# Connection Pooling

---



- **Hibernate besitzt Mechanismus für das Connection Pooling**
  - **für die Entwicklung und das Testen gut genug ist, aber nicht (unbedingt) für eine Produktionsumgebung**
  - **C3P0 ist für Produktionsumgebung eine beliebte produktionsreife Connection-Pooling-Bibliothek, die sehr einfach mit Hibernate zu verwenden ist. (<https://www.mchange.com/projects/c3p0>)**
  - **Konfiguration in persistence.xml, etwa Eigenschaftswerte wie minimale/maximale Anzahl von Verbindungen im Pool, Timeout-Werte usw. angeben**
  - **Rest erledigt Hibernate**
-

# Connection Pooling

---



## Java SE

- **Spezielle Konfigurationen nötig**

## Java EE / Spring:

- **Pooling wird vom Server unter Verwendung von DataSource durchgeführt**
  - **Verweis im persistence.xml auf DataSource statt DB-Connection**
-

# Connection Pooling

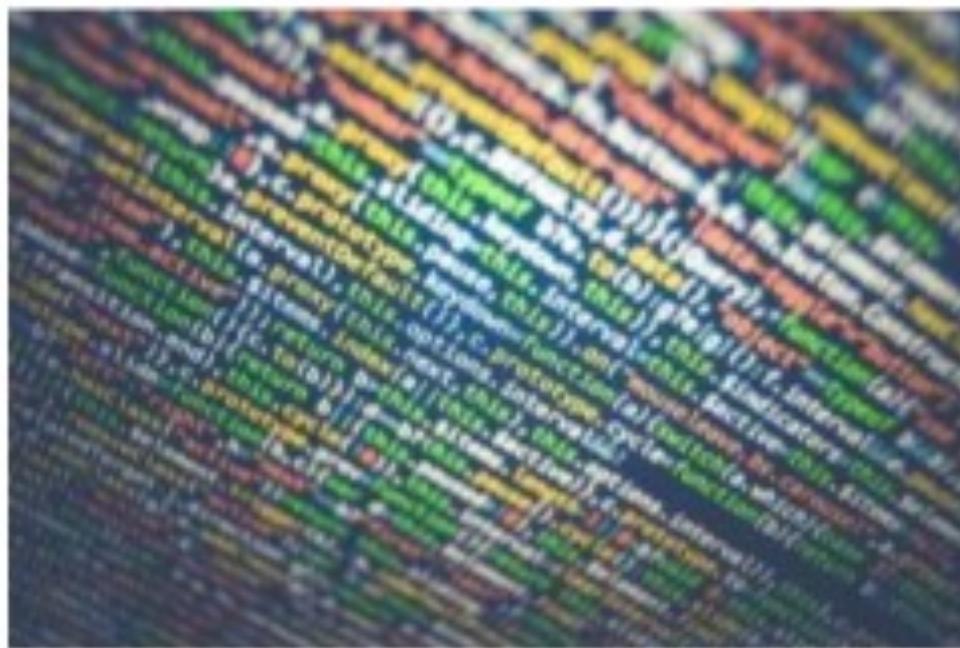


```
<persistence-unit name="jpa-connection-pool-example" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
        <!-- Configuring JDBC properties -->
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
        ...
        <!-- Hibernate properties -->
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        ...
        <!-- Configuring Connection Pool -->
        <property name="hibernate.c3p0.min_size" value="5" />
        <property name="hibernate.c3p0.max_size" value="20" />
        <property name="hibernate.c3p0.timeout" value="500" />
        <property name="hibernate.c3p0.max_statements" value="50" />
        <property name="hibernate.c3p0.idle_test_period" value="2000" />
    </properties>
</persistence-unit>
```



# Typische Schritte



## Typische Schritte (Recap)

---



1. Per Bootstrapping oder Injection erhalten wir eine EntityManagerFactory
  2. Ein EntityManagerFactory erzeugt einen EntityManager. In JPA wird eine Datenbankverbindung durch den EntityManager repräsentiert.
  3. Operationen, die den Datenbankinhalt verändern, müssen innerhalb einer Transaction-Instanz ablaufen
  4. Aktionen ausführen
  5. Transaktion committen / rollbacken
  6. EntityManager schliessen
  7. EntityManagerFactory schliessen
-

# Typische Schritte (EntityManagerFactory & EntityManager)

---



```
// 1)
final EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("<pu-name>");

// 2)
final EntityManager entityManager = entityManagerFactory.createEntityManager();

try
{
    // 3 & 4 & 5)
    performJPAActions(entityManager);
}

finally
{
    // 6) und 7)
    entityManager.close();
    entityManagerFactory.close();
}
```

## Typische Schritte (Transaction)

---



```
private static void performJPAActions(final EntityManager entityManager)
{
    try
    {
        // 3
        entityManager.getTransaction().begin();

        // 4
        executeStatements(entityManager);

        // 5
        entityManager.getTransaction().commit();
    }
    catch (final Exception e)
    {
        e.printStackTrace();
        entityManager.getTransaction().rollback();
    }
}
```

## Typische Schritte (Entities und Aktionen)

---



```
private static void executeStatements(final EntityManager entityManager)
{
    final Movie movie1 = new Movie(1L, "XYZ Gelöste Rätsel", false, LocalDate.of(1977, 5, 18));
    final Movie movie2 = new Movie(2L, "Bames Jond", true, LocalDate.of(2017, 9, 9));

    entityManager.persist(movie1);
    entityManager.persist(movie2);

    final String jpql = "SELECT m FROM Movie m";
    final TypedQuery<Movie> typedQuery = entityManager.createQuery(jpql, Movie.class);
    typedQuery.getResultList().forEach(System.out::println);
}
```



---

# DEMO

**BasicStepsExample.java**  
**ConnectionPoolExample.java**



---

## Exercises Part 2

<https://github.com/Michaeli71/JPA-Workshop.git>





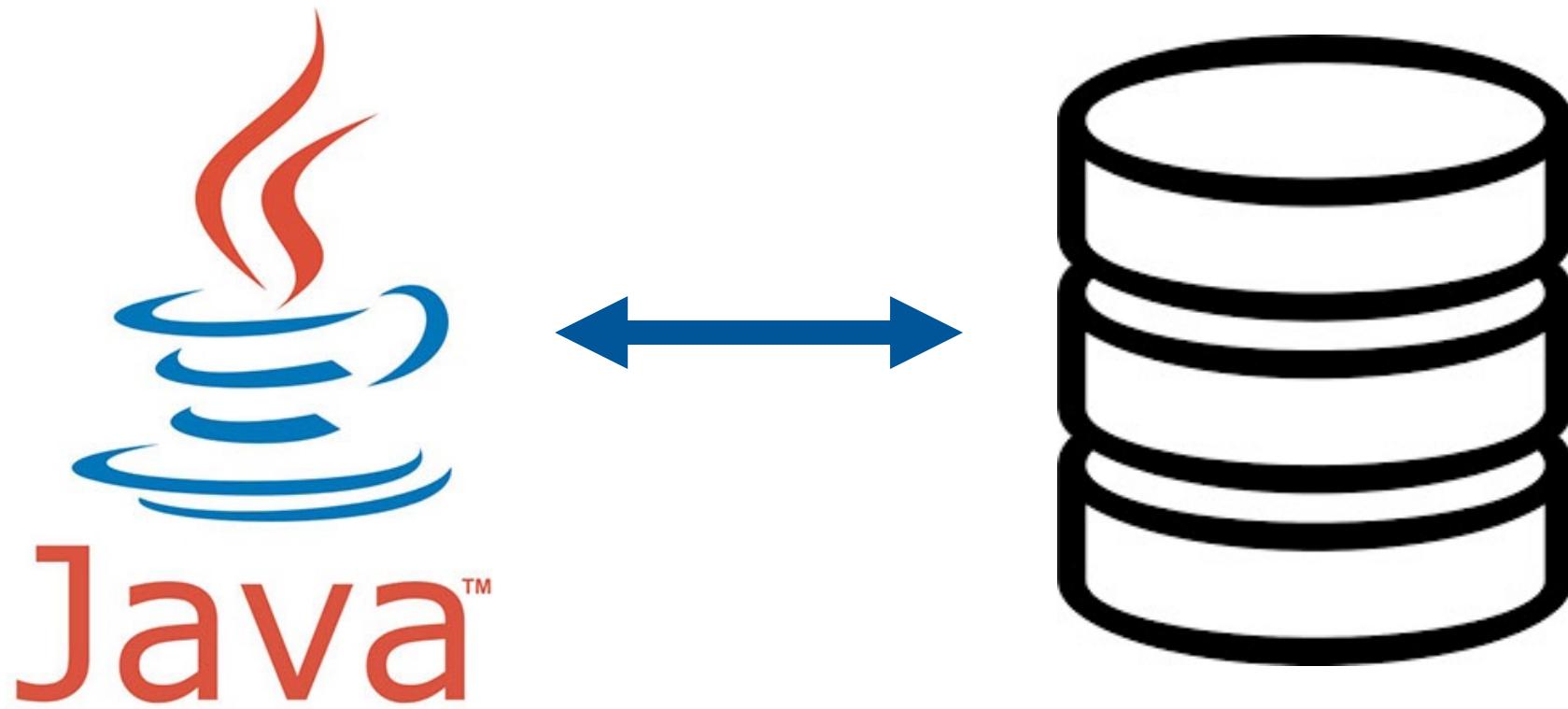
# PART 3: ORM Intro

## ORM = Object-Relational Mapping

---



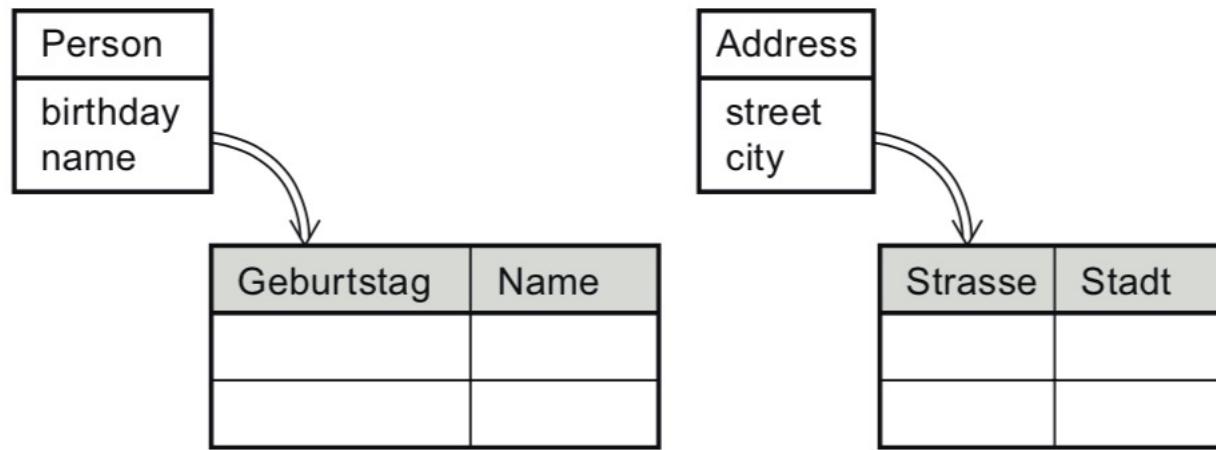
- **ORM = Object-Relational Mapping**
- **Abbildung vom objektorientierten Modell auf das Datenbankmodell**



# ORM = Object-Relational Mapping



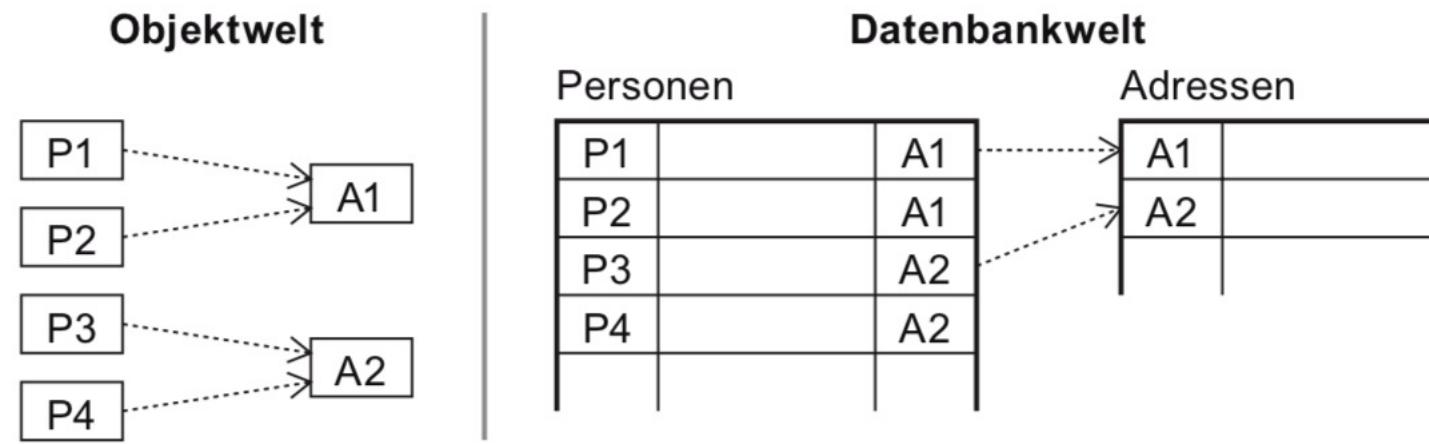
- Objekte auf Tabellen einer Datenbank abbilden:
  - Objekte in DB speichern



- Objekte aus DB rekonstruieren



- Objekte auf Tabellen einer Datenbank abbilden:
  - Beziehungen



- Vererbung
- ORM lässt sich mit etwas Mühe selbst programmieren.
- Wird aber doch mitunter komplex und herausfordernd (vor allem Assoziationen und Vererbung)



- **ORM mit «purem» JDBC ist recht aufwendig**
  - umso komplexer, aufwendiger und fehleranfälliger, je verzweigter der Objektgraph und je mehr Objekte und Assoziationen sowie Vererbungsbeziehungen existieren.
- **JPA erleichtert dies ungemein**
  - SQL-Anweisungen werden durch JPA automatisch generiert
  - Manchmal sind die generierten SQL-Kommandos suboptimal
  - Performance leidet, wenn viele referenzierte Daten verwaltet werden
- **Convention over Configuration**
  - Entitäten sind Java-Klassen mit Attributen und Annotations
  - In der Datenbank gibt es korrespondierende Tabellen und Spalten
  - **Implizite Abbildung erfolgt über Namen**
  - Explizite Abbildung erfolgt über Annotationen möglich (oder XML)

# Entity Class Wissenswertes

---



- **Entity Klasse muss**
  - mit `@Entity` annotiert werden
  - Einen No-Arg-Konstruktor haben (`public` / `protected`)
  - Weitere Konstruktoren sind erlaubt
- **Weitere Einschränkungen / Randbedingungen: Entity Klasse**
  - darf nicht `final` sein
  - Weder Methoden noch Attribute dürfen `final` sein
  - Klasse muss eine **Top-Level-Klasse** sein, keine innere Klasse



- Jede Entität muss eindeutige Identität besitzen
- @Id markiert ein Attribut einer Entität als Primärschlüssel
- Primärschlüssel können
  - über JPA oder die Datenbank automatisch generiert werden (@GeneratedValue)
  - von der Applikation erzeugt werden (interessant für Testing)



- **Primitive Typen**
  - char, short, int, long, byte, float, double, boolean
- **SerializableTypes**
  - Strings: `java.lang.String`
  - Primitive Wrappers: `Integer`, `Short`, `Long`, `Boolean`, `Double`, ...
  - `java.math.BigInteger`, `java.math.BigDecimal`
  - Java temporal types : `java.util.Date`, `java.util.Calendar`
  - JDBC temporal types: `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`
  - Java 8 temporal types: `java.time.LocalDate`, `LocalTime`, `LocalDateTime`
  - Byte-and character arrays: `byte[]`, `char[]`, `Byte[]`, `Character[]`
- **Enums**
- **Entities & Collections von Entities**
  - Collection, Set, List, Map

## Beispiel Entity Annotations an den Attributen



```
@Entity  
@Table(name = "PersonenJPA")  
public class Person implements Serializable  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    @Column(name = "Vorname")  
    private String firstName;  
  
    @Column(name = "Name")  
    private String lastName;  
  
    @Column(name = "Geburtstag")  
    private LocalDate birthday;  
  
    ...  
}
```



## Beispiel Entity Variante Annotations an den Methoden



```
@Entity  
@Table(name = "PersonenJPA")  
public class Person implements Serializable  
{  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private LocalDate birthday;  
  
    @Id  
    @GeneratedValue  
    public Long getId() // read only  
    {  
        return id;  
    }  
  
    @Column(name = "Vorname")  
    public String getFirstName()  
    {  
        return firstName;  
    }  
  
    ...  
}
```



## Entity Class Important Annotations (RECAP)

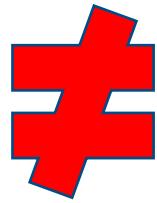
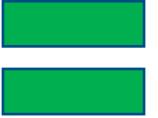
---



- **@Entity** -- Gibt an, dass die Klasse eine Entität ist
  - **@Table** -- Angabe der verwendeten primären Datenbanktabelle
  - **@Id** -- Angabe des Primärschlüsselfeldes
  - **@GeneratedValue** -- Angabe der Primärschlüssel-Generierungsmethode
  - **@Column** -- Angabe der gemappten Spalte für eine persistente Eigenschaft
  - **@Transient** -- Angabe von Feldern, die nicht persistent sind
  - **@Enumerated** -- Angabe des Mappings von Aufzählungen (ORDINAL / STRING)
-



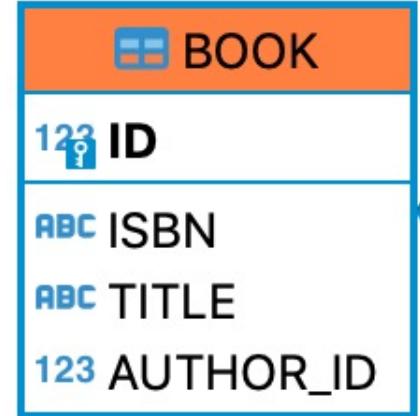
# Entity Equality



# Entity Equality



```
@Entity  
public class Book  
{  
    @Id  
    @GeneratedValue  
    long id;  
  
    String title;  
    String isbn;  
  
    @ManyToOne(fetch = FetchType.LAZY)  
    Author author;
```



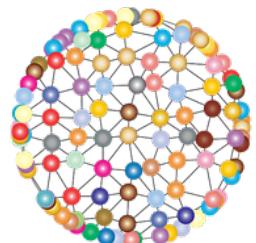


```
@Override  
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;
```

```
    Book other = (Book) obj;  
    return Objects.equals(author, other.author) && id == other.id &&  
          Objects.equals(isbn, other.isbn) &&  
          Objects.equals(title, other.title);  
}
```

```
@Override  
public int hashCode() {  
    return Objects.hash(author, id, isbn, title);  
}
```

BOOK	
123	ID
ABC	ISBN
ABC	TITLE
123	AUTHOR_ID



**Was sind mögliche  
Probleme? Wie ist Identität  
in DB definiert?**

# Entity Equality

---



- Laut Methodenkontrakten sollten nur *unveränderliche* Attribute für hashCode() genutzt werden
  - Alle Attribute im hashCode() führt aber zu Änderungen im Hash-Wert
  - Da bei JPA Attribute per Reflection gesetzt werden, immer getter und setter vorhanden, somit Immutability nicht zu erreichen
  - Außerdem: Im Kontext von Entities sind die Ids aber erst nach dem Persistieren vorhanden
  - Wenn primitive Typen genutzt werden, dann ist der Defaultwert 0 und nicht null, somit ist Unterscheidbarkeit noch schwieriger
  - Es gibt einige Hürden und
  - Es sind Kompromisse nötig
-

# Entity Equality



- **equals() und hashCode() basierend auf ID**

```
@Override  
public boolean equals(Object obj)  
{  
    if (this == obj)  
        return true;  
    if (obj == null || getClass() != obj.getClass())  
        return false;  
  
    Book other = (Book) obj;  
    return Objects.equals(id, other.id);  
}  
  
@Override  
public int hashCode()  
{  
    return Objects.hash(id);  
}
```



---

# Date and Time Integration



## JPA 2.2: Date and Time Support

---



- Weitere Infos auf: <https://thorben-janssen.com/map-date-time-api-jpa-2-2/>

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;

    private LocalDate date;

    private LocalTime time;

    private LocalDateTime dateTime;

    private OffsetTime offsetTime;

    private OffsetDateTime offsetDateTime;

    ...
}
```

## JPA 2.1: LocalDate & more => AttributeConverter



```
@Converter(autoApply = true)
public class LocalDateConverter implements AttributeConverter<LocalDate, String>
{
    @Override
    public String convertToDatabaseColumn(final LocalDate date)
    {
        if (date == null)
            return null;

        return date.toString();
    }

    @Override
    public LocalDate convertToEntityAttribute(final String value)
    {
        if (value == null)
            return null;

        return LocalDate.parse(value);
    }
}
```



# DEMO

**PersonDateAndTimeExample.java**

---



---

# Embeddable Classes



# Embeddable-Klassen

---



- **Embeddable-Klassen sind benutzerdefinierte, persistente Klassen, die als Wertetypen fungieren. Wie bei anderen Nicht-Entity-Typen können Instanzen einer Embeddable-Klasse nur als eingebettete Objekte in der Datenbank gespeichert werden, d. h. als Teil eines enthaltenden Entity-Objekts.**
  - **Eine Klasse wird als embeddable deklariert, indem sie mit der Annotation Embeddable gekennzeichnet wird.**
  - **Instanzen von Embeddable-Klassen sind immer in andere Entitätsobjekte eingebettet und erfordern keine separaten Speicher- und Abrufvorgänge. Daher kann die Verwendung von Embeddable-Klassen etwas Platz in der Datenbank sparen und die Effizienz verbessern.**
-

# Embeddable-Klassen

---



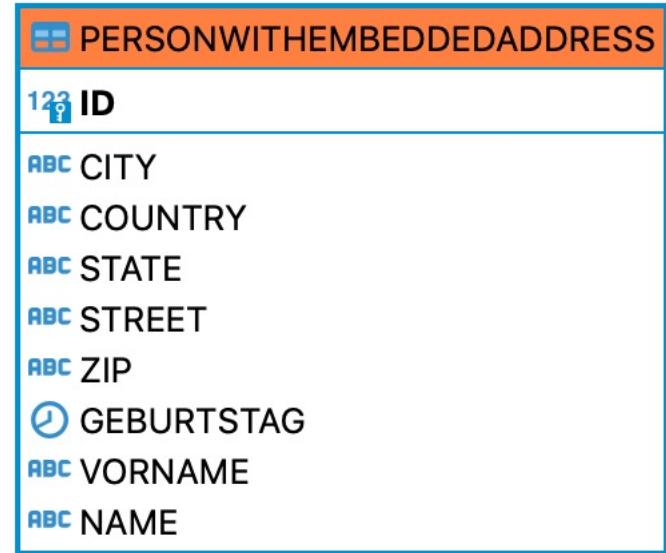
```
@Embeddable  
public class Address  
{  
    String street;  
    String city;  
    String state;  
    String country;  
    String zip;  
}
```

- **Embeddable-Klassen haben keine eigene Identität (Primärschlüssel), was zu einigen Einschränkungen führt (z. B. können ihre Instanzen nicht von verschiedenen Entitätsobjekten gemeinsam genutzt werden und sie können nicht direkt abgefragt werden)**
-

# «Eingebundene» Embeddable-Klassen



```
@Entity  
@Table(name = "PersonWithEmbeddedAddress")  
public class PersonWithAddress implements Serializable  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
    @Column(name = "Vorname")  
    private String firstName;  
    @Column(name = "Name")  
    private String lastName;  
    @Column(name = "Geburtstag")  
    private LocalDate birthday;  
  
    @Embedded  
    private Address address;  
  
    ...  
}
```





# DEMO

**PersonEmbeddedAddressExample.java**

---



---

# Enums und Element Collections



# Enums



```
@Entity(name = "EC_SimpleEmployee")
public class SimpleEmployee
{
    @Id
    @GeneratedValue
    private int id;
    private String name;
    private long salary;

    private Status status = Status.HIRED;
    private Hobbies hobby = Hobbies.Chess;
```

```
public enum Status {
    FIRED, HIRED, APPROVED, REJECTED;
}
```

```
public enum Hobbies {
    Soccer, Skating, Movies, Programming, Hiking, Music, HomeCinema, Reading, Chess
}
```

EC_SIMPLEEMPLOYEE
123 ID
123 HOBBY
ABC NAME
123 SALARY
123 STATUS

# Enums

---

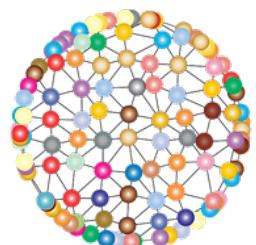


```
@Entity(name = "EC_SimpleEmployee")
public class SimpleEmployee
{
    @Id
    @GeneratedValue
    private int id;
    private String name;
    private long salary;

    @Enumerated(EnumType.ORDINAL)
    private Status status = Status.HIRED;

    @Enumerated(EnumType.STRING)
    private Hobbies hobby = Hobbies.Chess;
```

EC_SIMPLEEMPLOYEE	
123	ID
ABC	HOBBY
ABC	NAME
123	SALARY
123	STATUS



**Wie bilden wir aber eine  
Menge von Hobbies ab?**

# Listen und Maps von NICHT-Entities

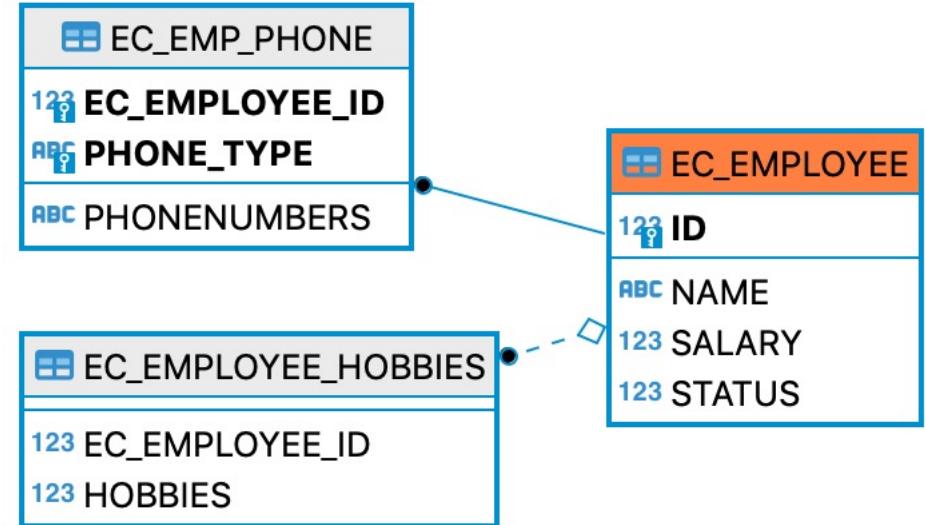


```
@Entity(name = "EC_Employee")
public class Employee
{
    @Id
    @GeneratedValue
    private int id;
    private String name;
    private long salary;

    private Status status = Status.HIRED;

    @ElementCollection
    private Set<Hobbies> hobbies = new HashSet<>();

    @ElementCollection
    @CollectionTable(name = "EC_EMP_PHONE")
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name = "PHONE_TYPE")
    private Map<PhoneType, String> phoneNumbers = new HashMap<>();
```



# Listen und Maps von NICHT-Entities

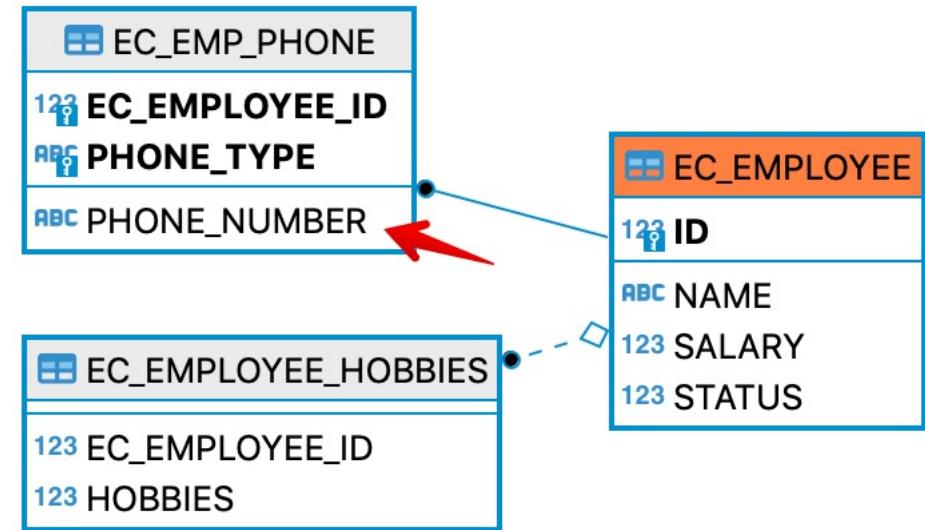


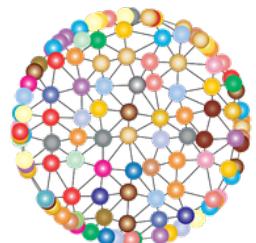
```
@Entity(name = "EC_Employee")
public class Employee
{
    @Id
    @GeneratedValue
    private int id;
    private String name;
    private long salary;

    private Status status = Status.HIRED;

    @ElementCollection
    private Set<Hobbies> hobbies = new HashSet<>();

    @ElementCollection
    @CollectionTable(name = "EC_EMP_PHONE")
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name = "PHONE_TYPE")
    @Column(name = "PHONE_NUMBER")
    private Map<PhoneType, String> phoneNumbers = new HashMap<>();
```





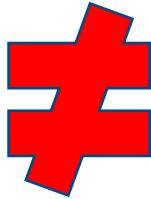
# Was sind Fallstricke bei ENUMs?

# Enums – Ordinal-Problem



```
public enum Status {  
    FIRED, HIRED, APPROVED, REJECTED;  
}
```

	123 ID	ABC HOBBY	ABC NAME	123 SALARY	123 STATUS
1	1	Programming	ENUM-PROBLEM-DEMO	0	3



```
public enum Status {  
    // FIRED,  
    HIRED, APPROVED, REJECTED;  
}
```

	123 ID	ABC HOBBY	ABC NAME	123 SALARY	123 STATUS
1	1	Programming	ENUM-PROBLEM-DEMO	0	2



# Primärschlüssel





- **Mögliche Typen für Primärschlüssel**
    - Primitive Typen: byte, int, short, long
    - Wrapper: Byte, Integer, Short, Long
    - java.lang.String
    - java.math.BigInteger
    - java.util.Date, java.sql.Date, java.time.LocalDate
  - **Bisher @Id in Kombination mit @GeneratedValue**
  - **VARIANTE: Selbstverwaltet mit get()- und set()-Methode**
-

# Objektidentität (Primary Keys) / ID-Generatoren (RECAP)



- Bisher @Id in Kombination mit @GeneratedValue

```
@Entity  
public class PlainAutomatic  
{  
    @Id  
    @GeneratedValue  
    long id; // set automatically  
  
    ...  
}
```



JPA bietet dabei folgende 4 verschiedene Möglichkeiten, Primärschlüsselwerte zu generieren:

- **AUTO**: JPA wählt die Generierungsstrategie basierend auf dem verwendeten Dialekt (oftmals **SEQUENCE**)
- **IDENTITY**: JPA verlässt sich auf eine automatisch inkrementierte Datenbankspalte, um den Primärschlüssel zu generieren.
- **SEQUENCE**: JPA generiert den Primärschlüsselwert basierend auf einer Datenbanksequenz.
- **TABLE**: JPA verwendet eine Datenbanktabelle statt einer Sequenz
- **Stärken/Schwächen**:  
<https://thorben-janssen.com/jpa-generate-primary-keys/>

# Primary Keys / ID-Generatoren

---



```
@Entity  
public class EntityWithAutoId1  
{  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    long id;  
}
```

```
@Entity  
public class EntityWithIdentityId  
{  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    long id;  
}
```

# Primary Keys / ID-Generatoren



```
@Entity
// Define a sequence - might also be in another class:
@SequenceGenerator(name = "seq", initialValue = 1, allocationSize = 100)
public class EntityWithSequenceId
{
    // Use the sequence that is defined above:
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq")
    @Id
    long id;
}

@Entity
@TableGenerator(name = "tab", initialValue = 0, allocationSize = 50)
public class EntityWithTableId
{
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "tab")
    @Id
    long id;
}
```

# Primary Keys / ID-Generatoren

---



- Wenn ein Primärschlüsselfeld **nicht mit @GeneratedValue annotiert ist, wird auch nicht automatisch ein Primärschlüsselwert generiert!**
- Die Anwendung ist dafür verantwortlich, einen Primärschlüssel zu setzen.
- Dies muss vor dem Persistieren geschehen!

```
@Entity
public class ApplicationManagedId
{
    @Id
    long id; // must be initialized by the application

    public long getId()
    {
        return id;
    }

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



---

# Primärschlüssel (Compound Keys)



## Zusammengesetzte / komplexe Primärschlüssel

---



- **Setzen sich aus mehreren Spalten zusammen (und diese müssen aus den zuvor aufgeführten unterstützten Typen bestehen)**
- **Es muss eine spezielle ID-Klasse definiert werden, die die Primärschlüsselfelder modelliert.**
- **Zwei Strategien möglich**
  - **Basierend auf @Embeddable:**  
@EmbeddedId markiert komplexes Primärschlüsselfeld der Entity-Klasse;  
@Embeddable markiert Primärschlüsselklasse
  - **Basierend auf @IdClass**  
@IdClass definiert Primärschlüsselklasse; mehrere Felder der Entityklasse werden mit @Id markiert

# Komplexe Primärschlüssel



```
@Entity  
public class CompositeEmbeddableIdEntity  
{  
    @EmbeddedId  
    CompositeEmbeddableId compositeId;  
  
    String name;  
    int additionalValue;  
}  
  
@Embeddable  
public class CompositeEmbeddableId implements Serializable  
{  
    int departmentId;  
    long projectId;  
  
    // equals() + hashCode()  
}
```

<class>c\_orm\_intro.idgeneration.CompositeEmbeddableIdEntity</class>

COMPOSITEEMBEDDABLEIDENTITY
120 DEPARTMENTID
121 PROJECTID
123 ADDITIONALVALUE
ABC NAME

# Komplexe Primärschlüssel



```
@Entity  
@IdClass(CompositeId.class)  
public class CompositeIdEntity  
{  
    @Id  
    int departmentId;  
    @Id  
    long projectId;  
  
    String name;  
    int additionalValue;  
    ...  
}  
  
public class CompositeId implements Serializable  
{  
    int departmentId;  
    long projectId;  
    // equals() + hashCode()  
}  
  
<class>c_orm_intro.idgeneration.CompositeIdEntity</class>
```

COMPOSITEIDENTITY
DEPARTMENTID
PROJECTID
ADDITIONALVALUE
NAME



---

# CRUD-Operationen



## EntityManager-Methoden für CRUD (RECAP)

---



- **C** `persist(Object entity)` – Speichert die Entity in der Datenbank und überführt sie in einen vom EntityManager verwalteten Zustand.
  - **R** `find(Class<T> entityClass, Object primaryKey)` – Sucht basierend auf dem übergebenen Primärschlüssel nach einem Datensatz in der Datenbank und gibt diesen als verwaltete Entity zurück.
  - **U** `-/- bzw. Methoden auf Entities` – Modifikationen erfolgen im Objektmodell, also etwa durch Aufruf von `set()`-Methoden oder Änderungen an Attributen
  - **D** `remove(Object entity)` – Entfernt eine Entity aus dem Persistence Context und später aus der Datenbank (sofern zwischenzeitlich nicht noch `persist()` aufgerufen wird).
-

# CRUD-Operationen für Personen



```
private static void executeStatements(final EntityManager entityManager)
{
    final Person michael = new Person("Micha-Date", "Inden", LocalDate.of(1971, 2, 7));
    final Person werner = new Person("Werner-Date", "Inden", LocalDate.of(1940, 1, 31));
    final Person tim = new Person("Tim-Date", "Bötz", LocalDate.of(1971, 3, 27));

    // CREATE
    entityManager.persist(michael);
    entityManager.persist(werner);
    entityManager.persist(tim);

    // READ
    final Long id = werner.getId();
    final Person wernerFromDb = entityManager.find(Person.class, id);

    // UPDATE
    wernerFromDb.setFirstName("Dr. h.c. Werner");

    // DELETE
    entityManager.remove(tim);
}
```

# CRUD-Operationen für Personen



```
final Person michael = new Person("Micha-Date", "Inden", LocalDate.of(1971, 2, 7));
final Person werner = new Person("Werner-Date", "Inden", LocalDate.of(1940, 1, 31));
final Person tim = new Person("Tim-Date", "Bötz", LocalDate.of(1971, 3, 27));

// CREATE
entityManager.persist(michael);
entityManager.persist(werner);

// READ
final Long id = werner.getId();
final Person wernerFromDb = entityManager.find(Person.class, id);

// UPDATE
wernerFromDb.setFirstName("Dr. h.c. Werner");

// DELETE
entityManager.remove(tim);
```

Gitter	ID	GEBURTSTAG	VORNAME	NAME
	1	1971-02-07	Micha-Date	Inden
	2	1940-01-31	Dr. h.c. Werner	Inden



- Manchmal möchte man Änderungen am lokalen Zustand wieder zurücksetzen
- `refresh()` erlaubt den Reload mit den in der DB gespeicherten Werten
- Die im Speicher befindliche Entity bzw. deren Werte werden mit denen der Datenbank überschrieben
- So kann man zwischenzeitliche DB-Änderungen reintegrieren
- Führt zu `IllegalArgumentException`, falls Entity nicht **MANAGED** ist
- Führt zu `EntityNotFoundException`, falls Entity nicht mehr in DB



**Detached Entity-Objekte werden von keinem EntityManager verwaltet und sind in ihrer Funktionalität eingeschränkt:**

- Viele JPA-Methoden akzeptieren keine Detached-Objekte
- Die Navigation auf Detached-Objekten wird nicht unterstützt (außer die Assoziationen sind schon geladen)
- Änderungen an Detached-Objekte werden nicht in der Datenbank reflektiert gespeichert. Um das zu erreichen, muss ein Detached-Objekt wieder mit einem EntityManager per merge() verbunden werden.
- Detached-Objekte für Datentransfer nützlich, etwa zwischen Applikationsschichten, im GUI keine Managed Entity darstellen



# DEMO

**PersonCrudExample.java**

---



---

## Exercises Part 3 & 4 (Aufgabe 1)

<https://github.com/Michaeli71/JPA-Workshop.git>





# PART 4: More ORM



**Klassen stehen in Relation zueinander und verwenden bzw. referenzieren andere Klassen zu Zusammenarbeit. Man spricht von Assoziationen:**

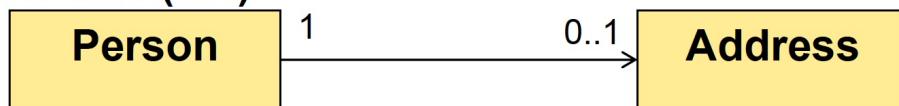
### 7 Formen von Beziehungen

- 1-zu-1 unidirektional (one-to-one unidirectional)
  - 1-zu-1 bidirektional (one-to-one bidirectional)
  - 1-zu-n unidirektional (one-to-many unidirectional)
  - 1-zu-n bidirektional (one-to-many bidirectional)
  - n-zu-1 unidirektional (many-to-one unidirectional)
  - m-zu-n unidirektional (many-to-many unidirectional)
  - m-zu-n bidirektional (many-to-many bidirectional)
-

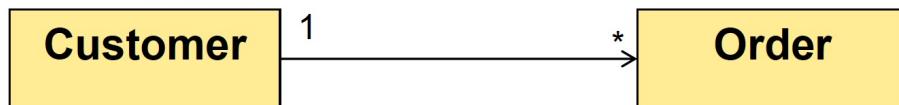


## Entity Relationships (unidirectional)

- One-To-One (1:1)



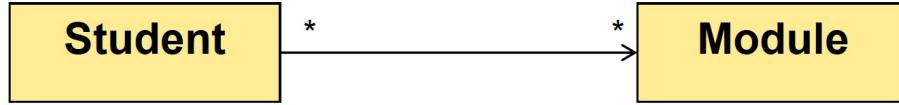
- One-To-Many (1:\*)



- Many-To-One (\*:1)



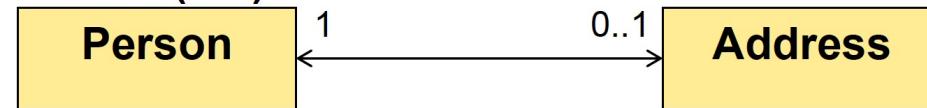
- Many-To-Many (\*:\*)



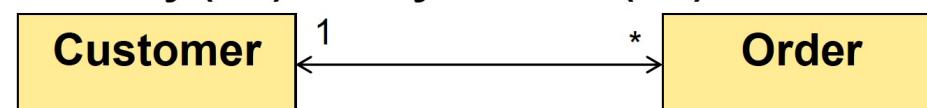


## Entity Relationships (bidirectional)

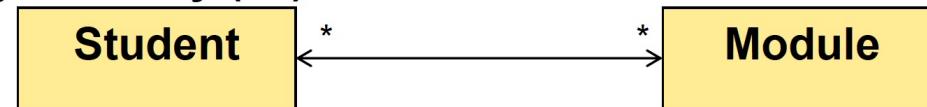
- One-To-One (1:1)



- One-To-Many (1:\*) / Many-To-One (\*:1)



- Many-To-Many (\*:\*)



# Assoziationen

---



- In JPA werden unidirektionale und bidirektionale Assoziationen unterstützt und über Annotations ausgedrückt:
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany
- Bei bidirektionalen Assoziationen muss eine Seite mit "mappedBy" gekennzeichnet werden

# Assoziationen unidirektional vs. bidirektional



- **Unidirektional (gerichtete Assoziation)**

```
@Entity  
public class Team  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @OneToMany  
    private Set<Person> teamMembers = new HashSet<>();
```

```
@Entity  
public class SimplePerson  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String firstName;  
    private String lastName;  
    private Date birthday;  
  
    // KEIN VERWEIS
```

- **in der SimplePerson-Entity-Klasse ist nichts zu Team definiert: Die SimplePerson weiß nichts von der Teammitgliedschaft. Deshalb erweitert JPA auch die SimplePerson-Tabelle nicht um eine Foreign-Key-Spalte, sondern muss eine separate Join-Tabelle hinzufügen.**

# Assoziationen unidirektional vs. bidirektional



- **Unidirektional (gerichtete Assoziation)**

```
@Entity  
public class Team  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @OneToMany  
    private Set<Person> teamMembers = ...
```

```
@Entity  
public class SimplePerson  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String firstName;  
    private String lastName;  
    private Date birthday;  
  
    // KEIN VERWEIS
```



# Assoziationen unidirektional vs. bidirektional



- **Bidirektional (zweiseitige Assoziation)**

```
@Entity  
public class Team  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @OneToMany  
    Set<Person> teamMembers = new HashSet<>();
```

```
@Entity  
public class SimplePerson  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String firstName;  
    private String lastName;  
    private Date birthday;  
  
    // VERWEIS AUF DAS TEAM  
    @ManyToOne  
    private Team team;
```

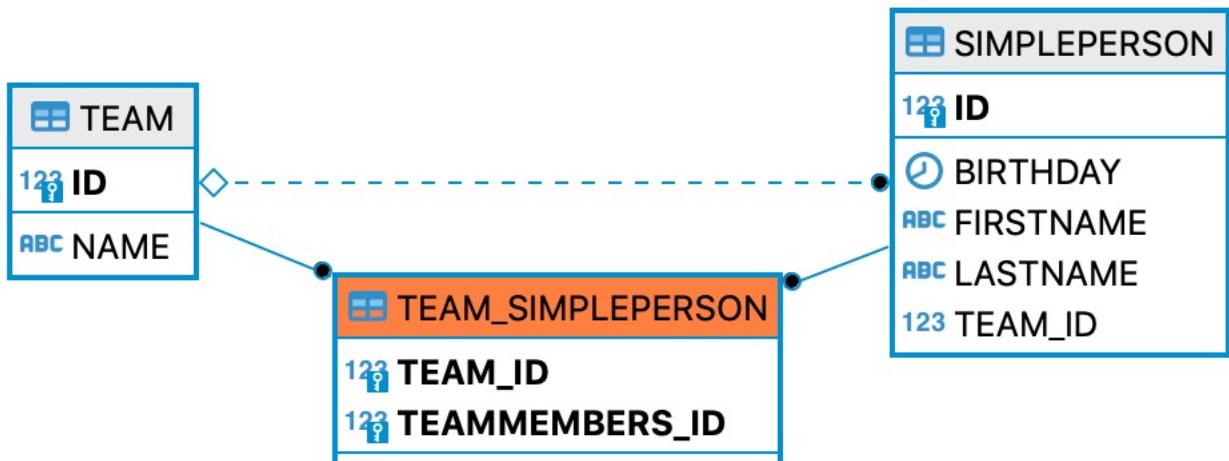
# Assoziationen unidirektional vs. bidirektional



- **Bidirektional (zweiseitige Assoziation)**

```
@Entity  
public class Team  
{  
    // ...  
  
    @OneToMany  
    Set<Person> teamMembers = new HashSet<>();
```

```
@Entity  
public class SimplePerson  
{  
    // ...  
  
    // VERWEIS AUF DAS TEAM  
    @ManyToOne  
    private Team team;
```



# Assoziationen unidirektional vs. bidirektional



```
@Entity  
public class Team  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @OneToMany(mappedBy="team")  
    Set<Person> teamMembers = new HashSet<>();
```

```
@Entity  
public class SimplePerson  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String firstName;  
    private String lastName;  
    private Date birthday;  
  
    // VERWEIS AUF DAS TEAM  
    @ManyToOne  
    private Team team;
```

# Assoziationen unidirektional vs. bidirektional



- Bidirektional (zweiseitige Assoziation) mit mappedBy

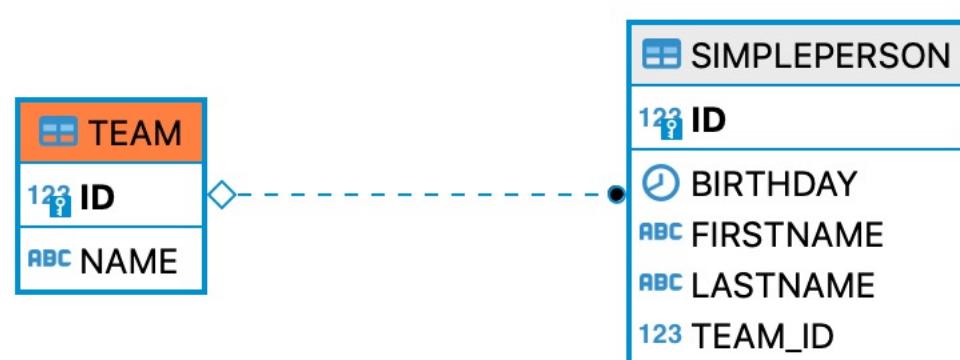
```
@Entity  
public class Team  
{
```

```
    @OneToMany(mappedBy="team")  
    Set<Person> teamMembers = new HashSet<>();
```

```
@Entity  
public class SimplePerson  
{
```

```
// VERWEIS AUF DAS TEAM  
    @ManyToOne  
    private Team team;
```

- Verbesserte bidirektionale Beziehung: Die SimplePerson-Tabelle wird um eine Foreign-Key-Spalte erweitert und es wird keine separate Join-Tabelle mehr benötigt:



## Fallstrick `toString()`

---



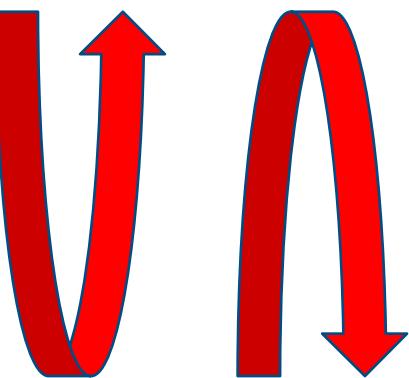
```
@Entity
public class SimplePerson
{
    @Override
    public String toString() {
        return "SimplePerson [id=" + id + ", firstName=" + firstName + ",
               lastName=" + lastName + ", birthday=" + birthday + ",
               team=" + team + "]";
    }
}
```

```
@Entity
public class Team
{
    @Override
    public String toString() {
        return "Team [id=" + id + ", name=" + name + ",
               teamMembers=" + teamMembers + "]";
    }
}
```

## Fallstrick `toString()`



```
@Entity  
public class SimplePerson  
{  
    @Override  
    public String toString() {  
        return "SimplePerson [id=" + id + ", firstName=" + firstName + ",  
               lastName=" + lastName + ", birthday=" + birthday + ",  
               team=" + team + "]";  
    }  
}
```



```
@Entity  
public class Team  
{  
    @Override  
    public String toString() {  
        return "Team [id=" + id + ", name=" + name + ",  
               teamMembers=" + teamMembers + "]";  
    }  
}
```

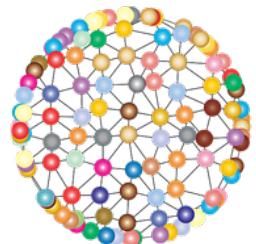
## Fallstrick `toString()`



```
@Entity
public class SimplePerson
{
    @Override
    public String toString() {
        return "SimplePerson [id=" + id + ", firstName=" + firstName + ",
               lastName=" + lastName + ", birthday=" + birthday + ",
               team=" + team.getName() + "]";
    }

    @Entity
    public class Team
    {
        @Override
        public String toString() {
            return "Team [id=" + id + ", name=" + name + ",
                   teamMembers=" + teamMembers + "]";
        }
    }
}
```

The diagram illustrates the inheritance relationship between the `SimplePerson` and `Team` classes. A green double-headed vertical arrow connects the two classes, indicating that `Team` inherits from `SimplePerson`. A large red double-headed vertical arrow connects the `SimplePerson` class to its `toString()` method, highlighting the overridden nature of the `toString()` method in `SimplePerson`.



**Ist die Modellierung mehrere  
Personen in jeweils nur  
einem Team korrekt?  
Sollte eine Person nicht in  
mehreren Teams sein?  
Gruppe != Team?**



# DEMO

**TeamAssociationsWithJpaExample.java**

---

# Auflösen von Beziehungen

---



- **Fetch-Strategie bestimmt, ob Beziehung während des Ladens aufgelöst wird**
  - **LAZY**: Beziehung wird nicht aufgelöst, sondern erst beim ersten Zugriff auf das Attribut
  - **EAGER**: Beziehung wird beim Lesen aufgelöst und ggf. einige Daten nachgeladen
- **Weitere Kontrollmöglichkeiten (später besprochen)**
  - **Entity Graph**: Es ist möglich, nur spezielle Attribute und Referenzierungen aufzulösen und zu laden. Dazu dient die Definition von sogenannten Entity Graphs, einer Beschreibung der benötigten Attribute der referenzierten Klassen (auch mehrschichtig).
  - **FETCH JOIN**: Bei Abfragen mit JPQL kann man das Fetch-Verhalten beeinflussen.



---

# Ladeverhalten FetchType & Kaskadierung CascadeType



## Entities laden und FetchType-Motivation

---



- **Problemkontext: Das Laden eines Entitätsobjekts aus der Datenbank kann das automatische Laden weiterer Entitätsobjekte verursachen.**
- **Teilweise wird das Laden automatisch über Assoziationen kaskadiert: Wenn ein Entity-Objekt geladen wird, werden mitunter auch alle Entity-Objekte geladen, die von diesem durch Navigation über diese Attribute erreicht werden können.**
- **Das kann unerwartet und ungewollt zu sehr viel geladen Daten führen.**
- **Theoretisch könnte es im Extremfall dazu führen, dass die gesamte Datenbank in den Speicher geladen wird.**
- **Deswegen ist eine Möglichkeit zur Steuerung nötig!**



- **Sowohl bei einzelnen Attributen als auch bei Assoziationen lässt sich das Ladeverhalten steuern:**
  - `fetch=FetchType.EAGER` -- die Daten werden sofort aus der Datenbank geladen.
  - `fetch=FetchType.LAZY` -- erlaubt dem JPA-Provider die Daten erst beim Zugriff und somit nur bei Bedarf aus der Datenbank zu laden, möglicherweise aber auch direkt
- **Defaults:**
  - Bei `@OneToOne` und `@ManyToOne` ist der Default **EAGER**
  - Bei `@OneToMany` und `@ManyToMany` ist der Default **LAZY**.

## Default FetchType modifizieren



- Abweichend vom Standard die referenzierten Manager NICHT laden:

```
@Entity  
class Employee  
{  
    // ...  
    @ManyToOne(fetch=FetchType.LAZY)  
    private Employee manager;  
    // ...  
}
```

- Abweichend vom Standard die referenzierten Projekte doch immer laden:

```
@Entity  
class Employee  
{  
    // ...  
    @ManyToMany(fetch = FetchType.EAGER)  
    private Collection<Project> projects;  
    // ...  
}
```

## Auswirkungen LAZY

---



- Zugriff auf “Manager“-Referenz löst (noch) keinen DB-Zugriff aus

```
Employee employee = entityManager.find(Employee.class, 1);
Employee employeeManager = employee.getManager(); // oft ein «leerer» Proxy
```

- Beim Zugriff auf Attribute wird die Entity dann automatisch im Hintergrund geladen:

```
String managerName = employeeManager.getName();
```

- Abfrage auf „Ladezustand“

```
PersistenceUtil util = Persistence.getPersistenceUtil();
boolean isObjectLoaded = util.isLoaded(employee);
boolean isAttributeLoaded = util.isLoaded(employee, "manager");
```

# Auswirkungen FetchType

---



- **Empfehlung:**
  1. Bevorzuge LAZY, fast immer die bessere Wahl
  2. Nutze dies oftmals auch für @OneToOne und @ManyToOne
- **Achtung:**
  - LAZY funktioniert nicht für DETACHED Entities
  - Wenn man dann Daten benötigt, kommt es zu undefiniertem Verhalten, oftmals Exceptions
  - Wenn man die Daten nach außen geben möchte, kann man einfach Collection.size() aufrufen

## Beispiel Kaskadierung

---



- Nehmen wir eine typische Business-Applikation mit Kunden und Adressen und Bestellungen und Bestellpositionen.
  - Es gibt zwei recht typische Anwendungsfälle für Kaskadierung im Kontext von Teile-Ganzes-Beziehungen:
    1. ERSTELLEN: Wenn ein Person- und eine Adresse-Objekt erzeugt wurden, dann müsste man normalerweise zweimal persist() aufrufen. Praktischerweise kann man assoziierte Objekte automatisch mit persistieren. Dadurch wird alles konsistent gespeichert.
    2. LÖSCHEN: Wird eine Bestellung gelöscht, so wäre es prima, wenn nicht alle einzelnen Bestellpositionen per Hand auch gelöscht werden müssten.
  - Standardmäßig werden keine(!) Operationen kaskadiert.
  - Muss expliziter angegeben werden.
-

# Kaskadierung

---



**Kaskadierung wird durch in der Assoziations-Annotation als Parameter cascade angegeben:**

```
@Entity  
class Employee  
{  
    // ...  
    @OneToOne(cascade=CascadeType.PERSIST)  
    private Address address;  
    // ...  
}  
  
@Entity  
class Order  
{  
    // ...  
    @OneToMany(cascade=CascadeType.REMOVE)  
    private List<OrderItems> orderItems;  
    // ...  
}
```

---

# JPA Cascade Types im Überblick

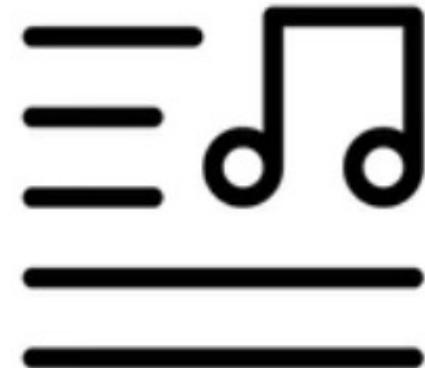
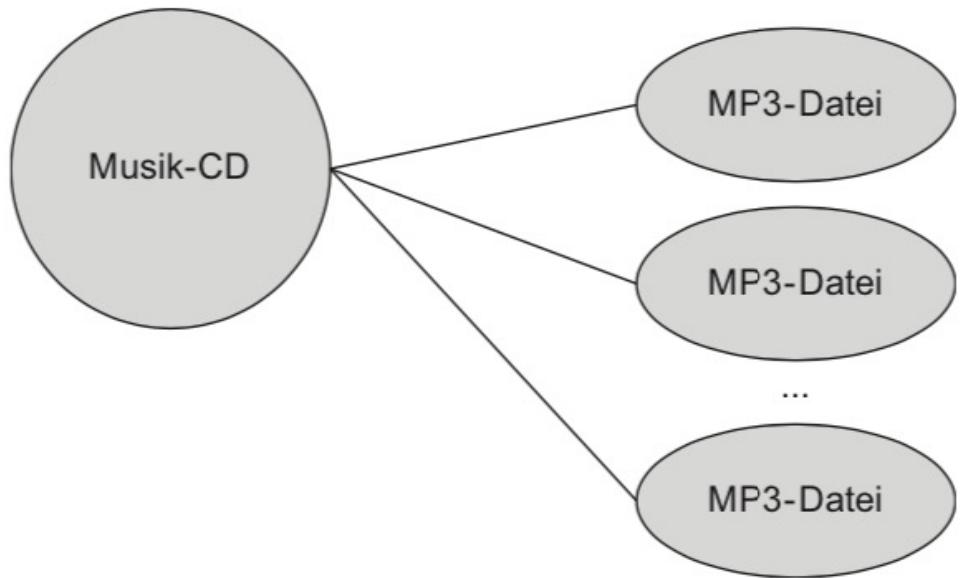
---



- **CascadeType.PERSIST** – persist()-Operationen wird auf verwandte Entitäten kaskadiert
  - **CascadeType.MERGE** -- verwandte Entitäten werden gemerged, wenn die besitzende Entität gemerged wird.
  - **CascadeType.REFRESH** -- die Operation refresh() wird auf verwandte Entitäten kaskadiert.
  - **CascadeType.REMOVE** – löscht automatisch alle verwandten Entitäten, wenn die besitzende Entity gelöscht wird.
  - **CascadeType.DETACH** – entfernt alle verbundenen Entitäten, wenn detach() erfolgt.
  - **CascadeType.ALL** – Kurzform für alle
- 
- **PERSIST** und **REMOVE** gebräuchlich, die anderen Varianten werden seltener benötigt.



- **Assoziation:** Verbindung zwischen zwei Entities
- **Aggregation:** Ganzes-Teile-Beziehungen / Master-Detail
- **Komposition:** enthaltene Teile können nicht ohne «Ganzes» existieren



- Beim Löschen dürfen beteiligte Objekte nur gelöscht werden, wenn die Teile ohne das Ganze nicht existieren (können/sollten).

# Orphan Removal

---



- Mit CascadeType.REMOVE kann man eine referenzierte Entity automatisch löschen, wenn die übergeordnete Entity gelöscht wird.
- JPA 2 unterstützt mit orphanRemoval das «agressivere Löschen»
- Bei orphanRemoval geht es um «verwaiste» Entities. Diese sollten aus der Datenbank gelöscht werden. Eine Entity gilt als verwaist, wenn diese nicht mehr mit ihrem übergeordneten Element verbunden ist, etwa getBooks().remove(7) => Buch «7» ist verwaist

```
@Entity  
class Employee  
{  
    // ...  
    // @OneToOne(cascade=CascadeType.REMOVE)  
    @OneToOne(orphanRemoval = true)  
    private Address address;  
    // ...  
}
```



**Worin liegt der  
Unterschied?**

# Orphan Removal

---



- Beim Löschen der «Eltern»-Entity, etwa Employee, wird auch die referenzierte Entity Address kaskadierend gelöscht.
- In dieser Hinsicht sind orphanRemoval=true und cascade=CascadeType.REMOVE identisch, und wenn orphanRemoval=true angegeben ist, ist CascadeType.REMOVE redundant.

## Unterschied:

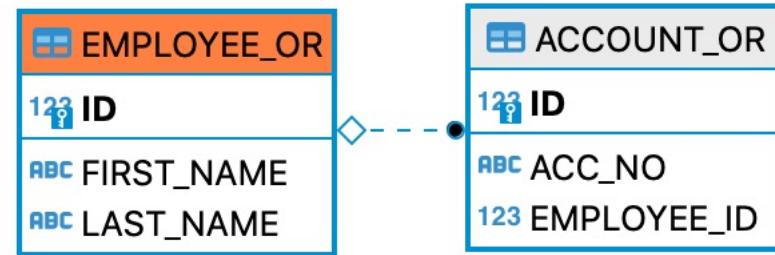
- Reaktion auf das Trennen einer Beziehung, etwa wenn das Adressfeld auf null oder auf ein anderes Address-Objekt gesetzt wird.
- Mit orphanRemoval=true wird die getrennte Address-Instanz automatisch entfernt. Dies ist nützlich, um abhängige Objekte (z. B. Address) zu bereinigen, die nicht ohne eine Referenz von einem Eigentümerobjekt (z. B. Employee) existieren sollten (**KOMPOSITION**)
- Wenn nur cascade=CascadeType.REMOVE angegeben ist, wird keine automatische Aktion durchgeführt, da das Trennen einer Beziehung kein Löschvorgang ist.

# Orphan Removal Beispiel



- "orphanRemoval = true" auf Account bedeutet im Wesentlichen, dass immer dann, wenn ein Account entfernt wird (also die Beziehung zwischen diesem Account und einem Employee gelöscht wird), dieser Account mit keinem anderen Mitarbeiter in der Datenbank verknüpft ist (d. h. verwaist), ebenfalls gelöscht werden soll.

```
@Entity  
@Table(name = "Employee_OR")  
public class EmployeeEntity implements Serializable  
{  
    @Id  
    @Column(name = "ID", unique = true, nullable = false)  
    @GeneratedValue  
    private Integer employeeId;  
    @Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)  
    private String firstName;  
    @Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)  
    private String lastName;  
  
    @OneToMany(orphanRemoval = true, mappedBy = "employee")  
    private Set<AccountEntity> accounts = new HashSet<>()
```



# Orphan Removal Beispiel



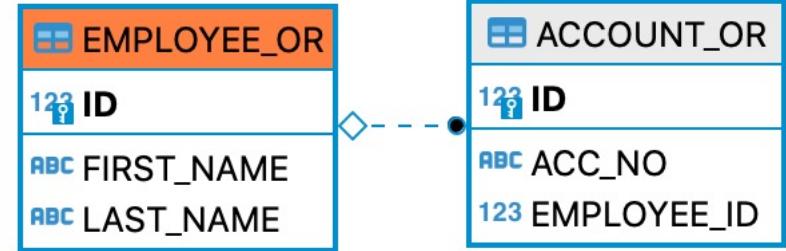
- **Probleme in der Regel durch Zugriffe auf Assoziationen**

```
public Set<AccountEntity> getAccounts() {  
    return accountEntities;  
}
```

```
public void setAccounts(Set<AccountEntity> accountEntities)  
{  
    this.accountEntities = accountEntities;  
}
```

- **Zweiseitige Referenzierung wird gerne mal vergessen:**

```
emp.getAccounts().add(acc1);  
emp.getAccounts().add(acc2);  
emp.getAccounts().add(acc3);  
acc1.setEmployee(emp);  
acc2.setEmployee(emp);  
acc3.setEmployee(emp);
```



# Orphan Removal Beispiel

---



- Durch Zugriffe wie folgt sind „Waisen“ möglich:

```
employee.getAccounts().remove(employee.getAccounts().iterator().next());  
employee.getAccounts().remove(accountToBeDeleted);
```

- Es wird dann nicht die andere Seite der Assoziation gelöscht!
- Abhilfe durch spezielle Zugriffsmethoden

```
public void addAccount(AccountEntity accountEntity) {  
    accountEntities.add(accountEntity);  
    accountEntity.setEmployee(this);  
}
```

```
public void removeAccount(AccountEntity accountEntity) {  
    accountEntities.remove(accountEntity);  
    accountEntity.setEmployee(null);  
}
```



---

# DEMO

**OrphanRemovalCascadeExample.java**

---



# Weitere Assoziationen

## 1:1 & n:m



# 1:1 -- Bidirektional



```
@Entity  
@Table(name = "simple_users")  
public class SimpleUser  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
    private String email;  
    private String password;  
  
    @OneToOne(mappedBy = "user",  
              fetch = FetchType.LAZY,  
              cascade = CascadeType.ALL)  
    private SimpleAddress address;
```

```
@Entity  
@Table(name = "simple_addresses")  
public class SimpleAddress  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String street;  
    private String city;  
    private String state;  
    private String zipCode;  
    private String country;  
  
    @OneToOne(cascade = CascadeType.PERSIST,  
              fetch = FetchType.LAZY)  
    @JoinColumn(name = "user_id",  
               nullable = false)  
    private SimpleUser user;
```

## 1:1 -- Bidirektional

---



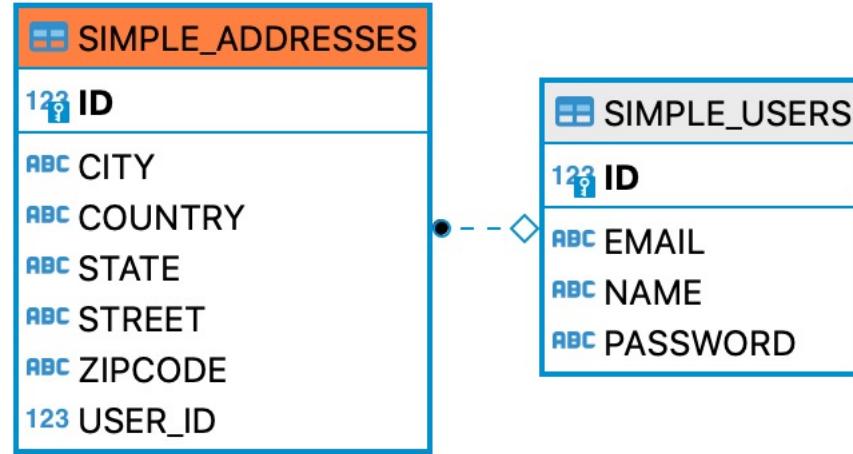
```
// create a user instance
SimpleUser user2 =
    new SimpleUser("Michael Inden", "michael_inden@hotmail.com", "0815");

// create an address instance
SimpleAddress address2 =
    new SimpleAddress("In der Ey 50", "Zürich", "Switzerland", "8047", "CH");

// Verknüpfen
user2.setAddress(address2);
address2.setUser(user2);

entityManager.persist(address2);

// nicht nötig durch cascade persist
// entityManager.persist(user2);
```



Es kann jedoch nur eine Zeile in SIMPLE\_ADDRESSES mit einer Zeile in SIMPLE\_USERS verknüpft sein

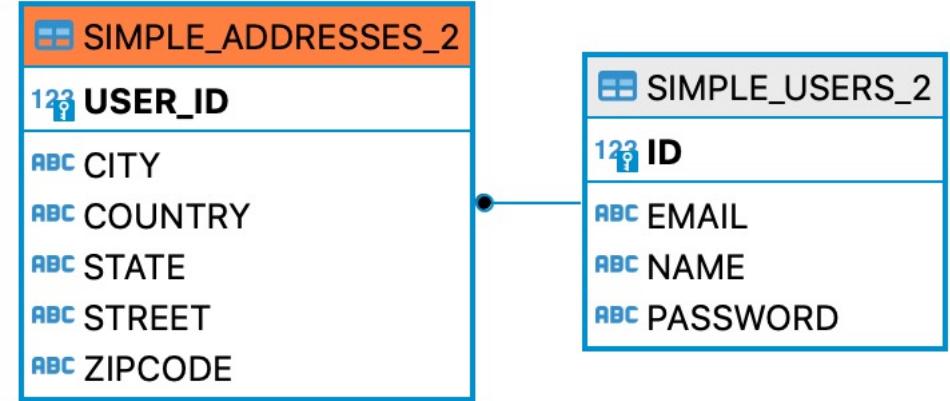
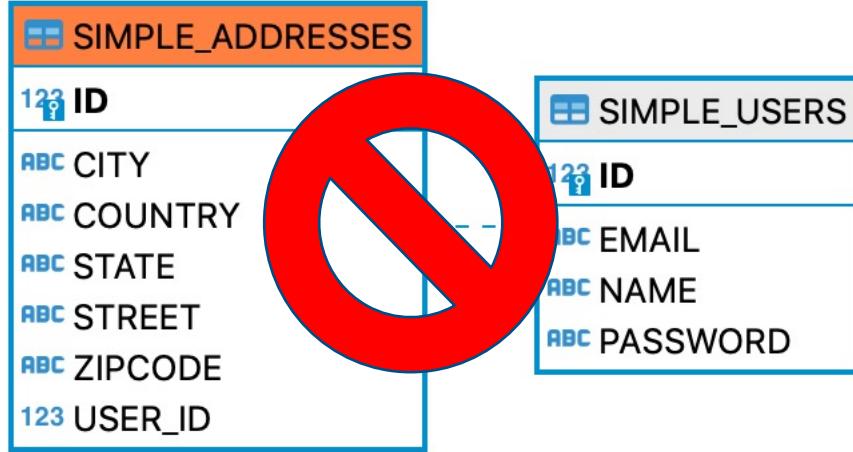
=> daher ist es sinnvoller, wenn die beide den gleich PK nutzen würden!

# 1:1 -- Bidirektional



```
@Entity  
@Table(name = "simple_users_2")  
public class SimpleUser2  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
    private String email;  
    private String password;  
  
    @OneToOne(mappedBy = "user",  
              fetch = FetchType.LAZY,  
              cascade = CascadeType.ALL)  
    private SimpleAddress2 address;
```

```
@Entity  
@Table(name = "simple_addresses_2")  
public class SimpleAddress2  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String street;  
    private String city;  
    private String state;  
    private String zipCode;  
    private String country;  
  
    @OneToOne(fetch = FetchType.LAZY)  
    @MapsId  
    @JoinColumn(name = "user_id",  
               nullable = false)  
    private SimpleUser2 user;
```



Durch diese Änderung ist der **SIMPLE\_ADDRESSES** PK auch der FK und beide Tabellen haben denselben PK.

Als Letztes: Wir brauchen nicht mal mehr eine bidirektionale Verbindung!

# 1:1 – Unidirektional mit MapId

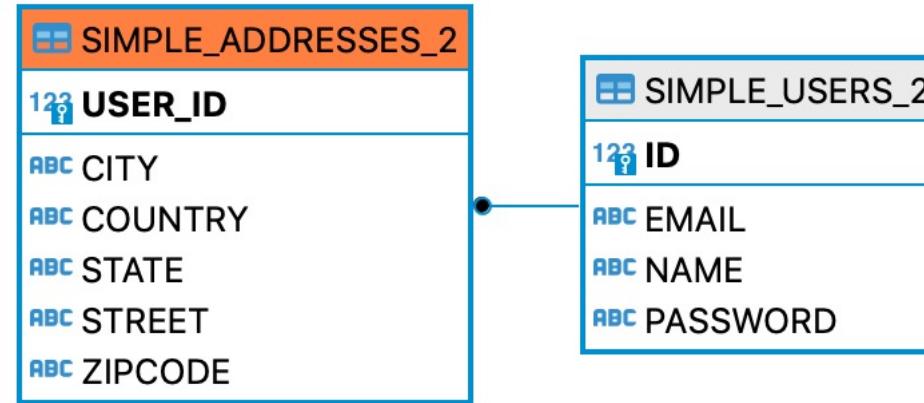


```
@Entity  
@Table(name = "simple_users_2")  
public class SimpleUser2  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String name;  
    private String email;  
    private String password;  
  
    @OneToOne(mappedBy = "user",  
              fetch = FetchType.LAZY,  
              cascade = CascadeType.ALL)  
    private SimpleAddress2 address;
```

```
@Entity  
@Table(name = "simple_addresses_2")  
public class SimpleAddress2  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String street;  
    private String city;  
    private String state;  
    private String zipCode;  
    private String country;  
  
    @OneToOne(fetch = FetchType.LAZY)  
    @MapsId  
    @JoinColumn(name = "user_id",  
               nullable = false)  
    private SimpleUser2 user;
```



Beide Tabellen besitzen den gleichen Schlüssel einmal als Primary Key und einmal als Foreign Key und zudem ist der FK == PK



Aufgrund des gleichen Schlüsselwerts brauchen nicht mal mehr eine bidirektionale Verbindung!

```
SimpleAddress2 retrievedAddress = entityManager.find(SimpleAddress2.class, userId);
System.out.println(retrievedAddress);
```



---

# DEMO

[One\\_to\\_one\\_Example.java](#)  
[One\\_to\\_one\\_MapsId\\_Example.java](#)

---



n : m





- Studenten und Kurse, jeweils @ManyToMany



```
@Entity  
class Student  
{  
    @Id  
    @GeneratedValue  
    Long id;  
  
    String name;  
  
    @ManyToMany(mappedBy = "likedBy")  
    Set<Course> likedCourses = new HashSet<>();
```

```
@Entity  
class Course  
{  
    @Id  
    @GeneratedValue  
    Long id;  
  
    String name;  
  
    @ManyToMany  
    Set<Student> likedBy = new HashSet<>();
```



- JOIN-Tabelle anpassen



```

@Entity
class Student
{
    @Id
    @GeneratedValue
    Long id;

    String name;

    @ManyToMany(mappedBy = "likedBy")
    Set<Course> likedCourses
        = new HashSet<>();
}
  
```

```

@Entity
class Course
{
    @Id
    @GeneratedValue
    Long id;

    String name;

    @ManyToMany
    @JoinTable(name = "courses_likes",
               joinColumns = @JoinColumn(name = "course_id"),
               inverseJoinColumns = @JoinColumn(name = "student_id"))
    Set<Student> likedBy = new HashSet<>();
}
  
```



# DEMO

[Students\\_And\\_Courses\\_Example.java](#)

---



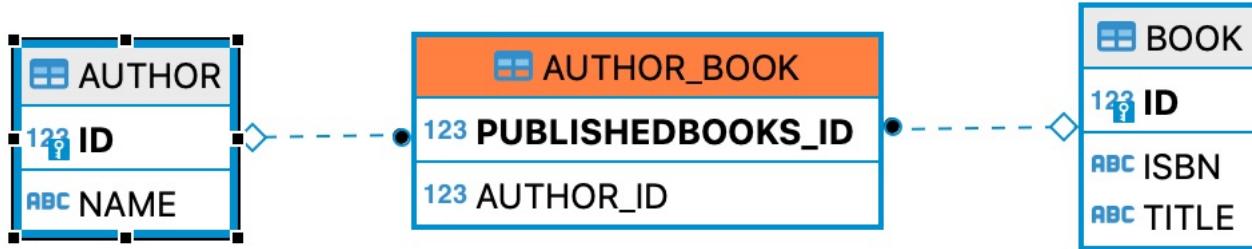
---

# Fallstricke / Wissenswertes / Performance uni-/bidirektionale Relationen

## 1:n (unidirektional => Join-Tabelle)



- 1 Author -> n Books



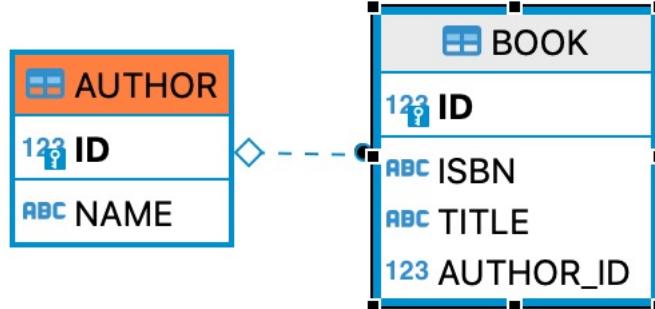
```
@Entity
public class Author
{
    @Id
    @GeneratedValue
    long id;
    String name;
    // 1:n
    @OneToMany
    List<Book> publishedBooks = new ArrayList<>();
```

```
@Entity
public class Book
{
    @Id
    @GeneratedValue
    long id;
    String title;
    String isbn;
    // @ManyToOne
    // Author author;
```

## n:1 (unidirektional => Join-Column, aber FetchType)



- n Books -> 1 Author



```
@Entity
public class Author
{
    @Id
    @GeneratedValue
    long id;
    String name;

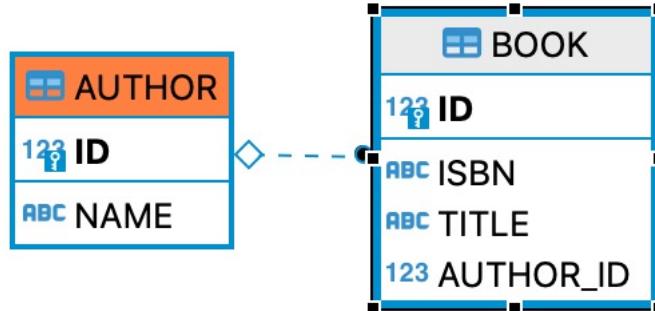
    //List<Book> publishedBooks = new ArrayList<>();
```

```
@Entity
public class Book
{
    @Id
    @GeneratedValue
    long id;
    String title;
    String isbn;
    @ManyToOne
    Author author;
```

## n:1 (unidirektonal => Join-Column)



- n Books -> 1 Author



```
@Entity
public class Author
{
    @Id
    @GeneratedValue
    long id;
    String name;

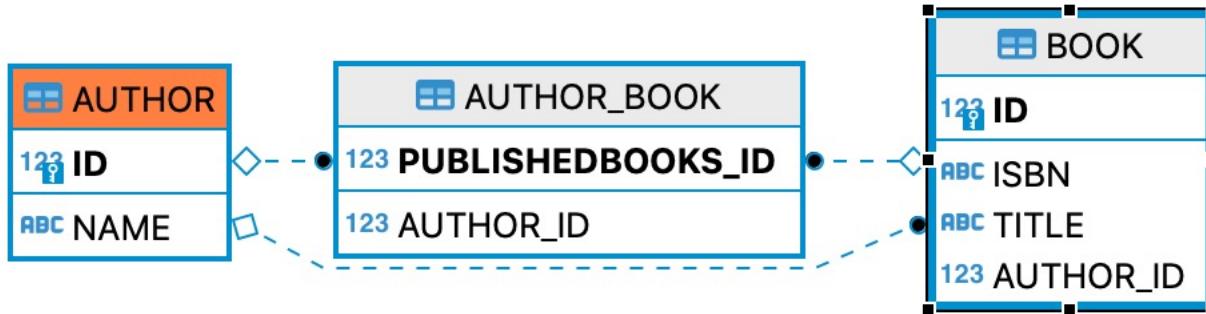
    //List<Book> publishedBooks = new ArrayList<>();
```

```
@Entity
public class Book
{
    @Id
    @GeneratedValue
    long id;
    String title;
    String isbn;
    @ManyToOne(fetch = FetchType.LAZY)
    Author author;
```

## 1:n <-> n:1(bidirektional ohne mappedBy)



- 1 Author <-> n Books



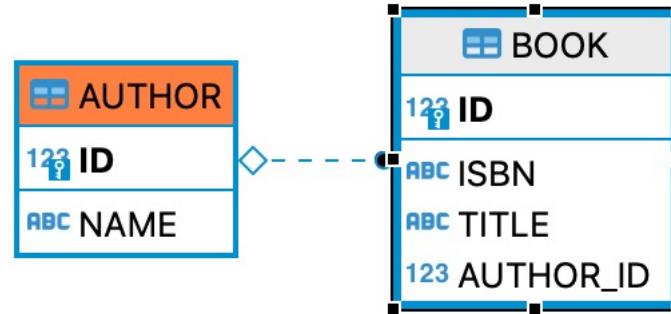
```
@Entity
public class Author
{
    @Id
    @GeneratedValue
    long id;
    String name;
    // 1:n
    @OneToMany
    List<Book> publishedBooks = new ArrayList<>();
```

```
@Entity
public class Book
{
    @Id
    @GeneratedValue
    long id;
    String title;
    String isbn;
    @ManyToOne(fetch = FetchType.LAZY)
    Author author;
```

## 1:n <-> n:1 (bidirektional mit @ManyToOne)



- 1 Author <-> n Books



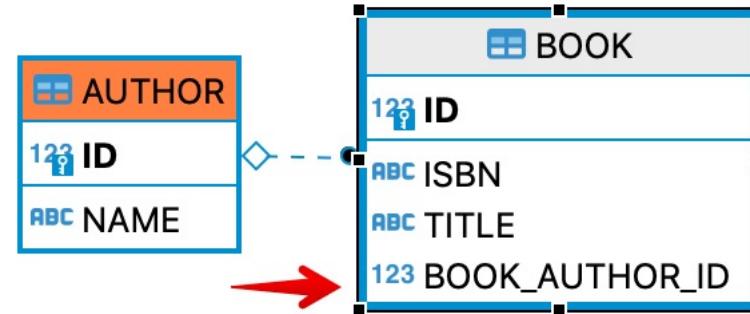
```
@Entity  
public class Author  
{  
    @Id  
    @GeneratedValue  
    long id;  
    String name;  
    // 1:n  
    @OneToMany(mappedBy = "author")  
    List<Book> publishedBooks = new ArrayList<>();
```

```
@Entity  
public class Book  
{  
    @Id  
    @GeneratedValue  
    long id;  
    String title;  
    String isbn;  
    @ManyToOne(fetch = FetchType.LAZY)  
    Author author;
```

## 1:n <-> n:1 (bidirektional)



- 1 Author <-> n Books



```
@Entity  
public class Author  
{  
    @Id  
    @GeneratedValue  
    long id;  
    String name;  
    // 1:n  
    @OneToMany(mappedBy = "author")  
    List<Book> publishedBooks = new ArrayList<>();
```

```
@Entity  
public class Book  
{  
    @Id  
    @GeneratedValue  
    long id;  
    String title;  
    String isbn;  
    @ManyToOne(fetch = FetchType.LAZY)  
    @JoinColumn(name = "book_author_id")  
    Author author;
```

## @JoinColumn und mappedBy

---



- Die Annotations wie @ManyToOne, @OneToMany und @ManyToMany weisen JPA/Hibernate an, ein Mapping zu erstellen. **Standardmäßig wird dies über eine separate Join-Tabelle** durchgeführt.
- **@JoinColumn** – Damit wird eine **Join-Column** erstellt, wenn noch keine vorhanden ist. In beiden Fällen kann die Annotation verwendet werden, um die Join-Spalte zu benennen.
- **mappedBy** – Hiermit wird JPA angewiesen: Erstelle KEINE weitere Join-Tabelle, da die Beziehung bereits von der gegenüberliegenden Entität dieser Beziehung abgebildet wird.

## Fazit

---



- Vermeide unidirektional 1:n (@OneToMany)
  - Vermeide bidirektional ohne mappedBy
  - Bevorzuge @ManyToOne, aber Achtung FetchType.EAGER als default
  - Bevorzuge @ManyToOne mit FetchType.LAZY
  - Bevorzuge mappedBy: So wird keine Join-Tabelle erzeugt
-



# DEMO

**Author\_and\_Books\_Example.java**

---



# Vererbung

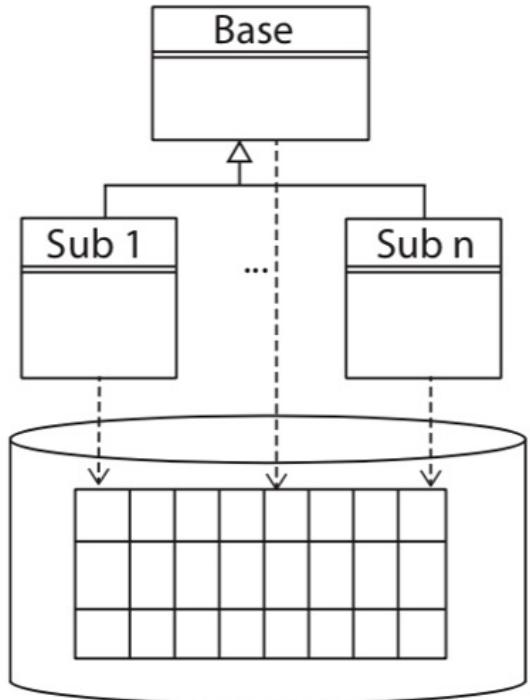


# Vererbungshierarchien

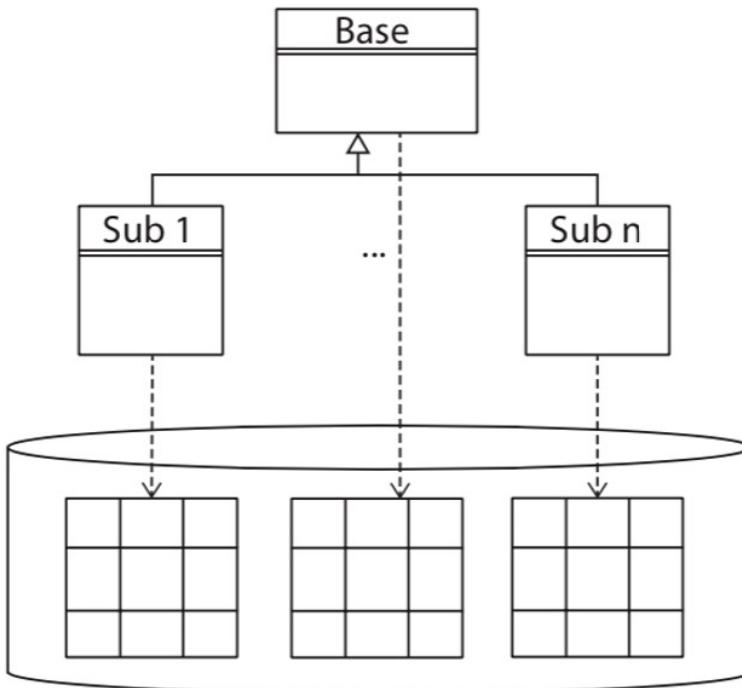


- Mit JPA lassen sich Vererbungshierarchien abbilden
- Alle Klassen der Vererbungshierarchie müssen eine Entity sein
- In `javax.persistence.InheritanceType` diese Konstanten definiert:

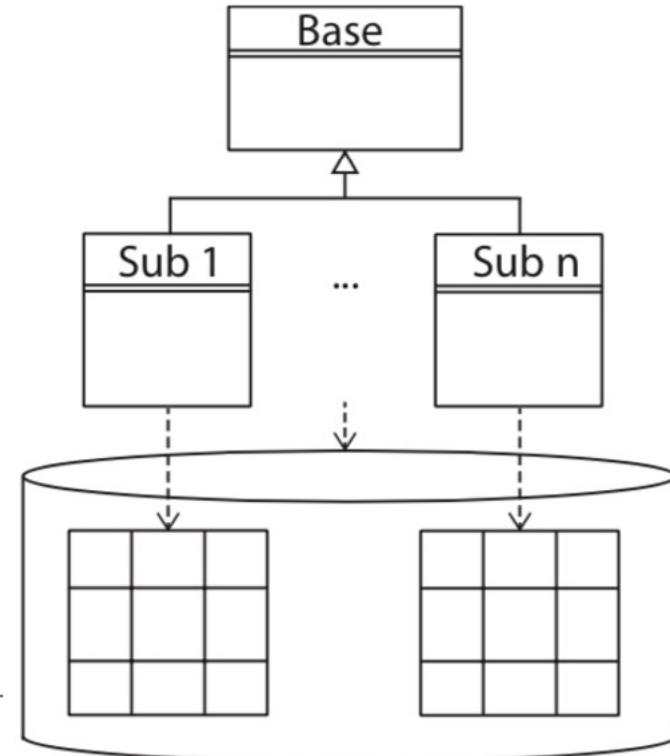
SINGLE\_TABLE (Standard)



JOINED



TABLE\_PER\_CLASS



# Vererbungshierarchien

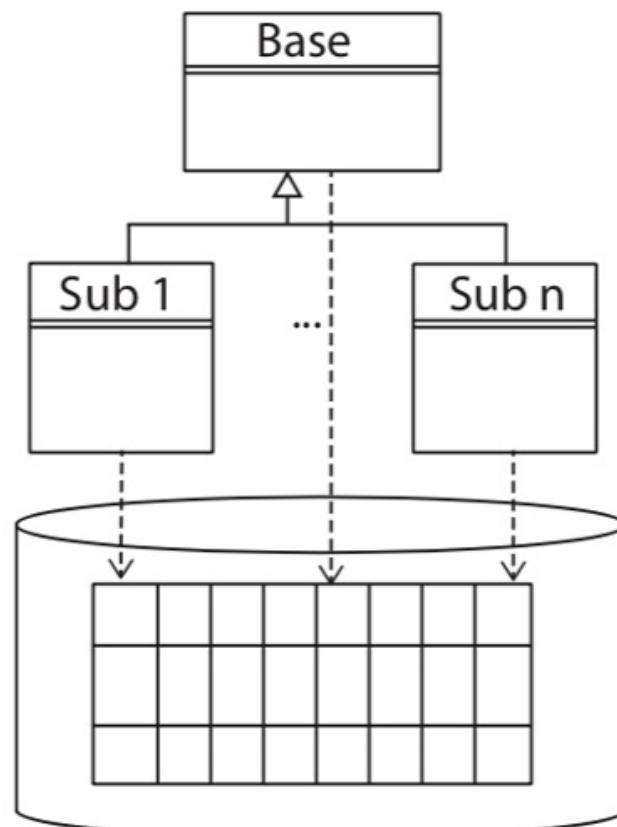


## »Single Table« – Eine Tabelle pro Vererbungshierarchie

Diese Form der Abbildung fasst alle Klassen einer Vererbungshierarchie in einer »One-For-All«-Tabelle oder »Single Table« zusammen, die sowohl die Attribute der Basisklasse als auch alle Attribute der Subklassen speichert.

Folgende Tabellenstruktur verdeutlicht den Aufbau für eine Klassenhierarchie mit einer Basisklasse **Base** und zwei Subklassen **Sub1** und **Sub2**. Die Spalten für die jeweiligen Attribute der Klassen sind hier symbolisch mit dem Klassennamen, gefolgt von den Postfixes **-1**, **-2** usw., dargestellt:

Base-1	Base-2	Sub1-1	Sub1-2	Sub1-3	Sub1-4	Sub2-1	Sub2-2	Typ
Müller	27	value1	value2	value3	value4	NULL	NULL	Sub1
Meyer	33	NULL	NULL	NULL	NULL	value1	value2	Sub2



# Vererbungshierarchien



```
@Entity  
@Inheritance(strategy =  
InheritanceType.SINGLE_TABLE)  
public class BaseProduct  
{  
    @Id  
    private long productId;  
    private String name;  
  
    // constructor, getters, setters  
}
```

```
@Entity  
public class Book extends BaseProduct  
{  
    private String author;  
  
    // ...  
}
```

```
@Entity  
public class Book extends BaseProduct  
{  
    private String author;  
  
    // ...  
}
```

# Vererbungshierarchien



Um ermitteln zu können, welcher Klasse ein Datensatz angehört, benötigt man – wie schon bei Mehrfachabbildungen – eine (rein technisch motivierte) zusätzliche Spalte im Datenbankmodell. Diese Spalte wird bekanntlich **Diskriminatorspalte** genannt. Im Beispiel ist dies die Spalte Typ. Verdeutlichen wir uns diese Variante an einer Basisklasse Person sowie den Subklassen Sportler (mit HF=Herzfrequenz) und Mitarbeiter:

Person	Person	Sportler	Sportler	Mitarbeiter	Mitarbeiter	Typ
Name	Alter	HF-MIN	HF-MAX	Abteilung	Titel	
<hr/>						
Müller	27	48	198	NULL	NULL	Sportler
Meyer	33	NULL	NULL	Entwicklung	Dr.	Mitarbeiter

# Vererbungshierarchien



```
@Entity  
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name="product_type",  
                     discriminatorType = DiscriminatorType.INTEGER)  
public class BaseProduct  
{  
    @Id  
    private long productId;  
    private String name;  
  
    // constructor, getters, setters  
}
```

```
@Entity  
@DiscriminatorValue("1")  
public class Book extends BaseProduct  
{  
    private String author;  
    // ...  
}
```

```
@Entity  
@DiscriminatorValue("2")  
public class Car extends BaseProduct  
{  
    private String color;  
    // ...  
}
```

# Vererbungshierarchien -- JOINED



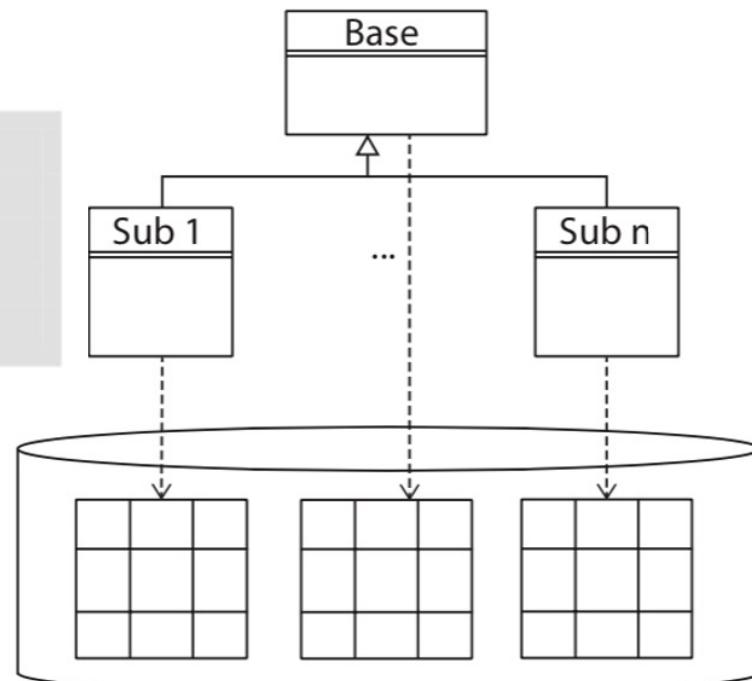
## »Joined« – Eine Tabelle pro Basis- und je eine pro Subklasse

Bei dieser Form der Abbildung wird für jede Klasse in einer Vererbungshierarchie eine eigenständige Tabelle genutzt. Die Attribute einer Klasse werden auf die Spalten der klassenspezifischen Tabelle abgebildet. Diese Variante ähnelt dem objektorientierten Gedanken, wo Subklassen nur die Unterschiede zu den jeweiligen Basisklassen beschreiben.

Für das Beispiel ergeben sich dann drei Tabellen. Beginnen wir mit der Tabelle Personen, die die Daten der Basisklasse Person speichert:

**Tabelle Personen**

ID	Name	Alter
1	Müller	27
2	Meyer	33



## Vererbungshierarchien -- JOINED



Zudem nutzen wir die beiden folgenden Tabellen Mitarbeiter und Sportler, die jeweils Spalten für die Attribute der korrespondierenden Subklassen besitzen:

### Tabelle Sportler

HF-MIN	HF-MAX	PersonenID
48	198	1

### Tabelle Mitarbeiter

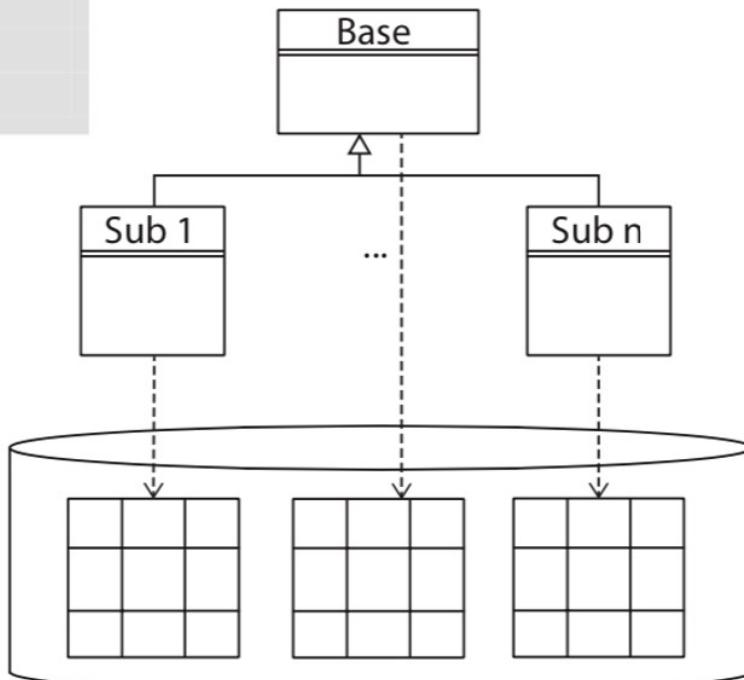
Abteilung	Titel	PersonenID
Entwicklung	Dr.	2

### Vorteile:

- Normalisiertes Schema, eine Datenbanktabelle für jede Klasse
- Alle Felder können mit Not-Null-Constraints definiert werden
- Fremdschlüsselreferenzen auf konkrete Unterklassen sind möglich

### Nachteile:

- Jeder Entity-Zugriff muss über mehrere Tabellen gehen
- Für tiefe Vererbungsbeziehungen potenziell (sehr) schlechte Performance

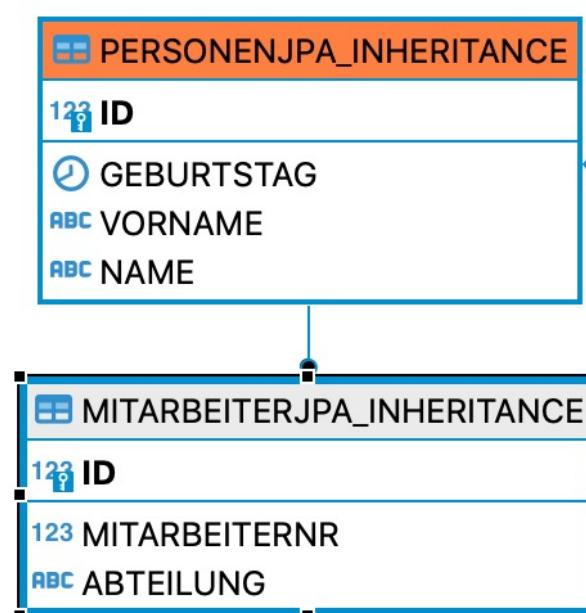


# Vererbungshierarchien



```
@Entity  
@Table(name = "PersonenJPA_Inheritance")  
@Inheritance(strategy=InheritanceType.JOINED)  
public class Person implements Serializable  
{  
    @Id  
    @GeneratedValue  
    private Long id;
```

```
@Entity  
@Table(name = "MitarbeiterJPA_Inheritance")  
public class Employee extends Person  
{  
    // KEINE ID, wird geerbt  
  
    @Column(name = "MitarbeiterNr")  
    private Integer employeeNumber;  
  
    @Column(name = "Abteilung")  
    private String workgroup;
```



# Vererbungshierarchien



```
@Entity  
@Table(name = "PersonenJPA_Inheritance")  
@Inheritance(strategy=InheritanceType.JOINED)  
public class Person implements Serializable  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    @Column(name = "Vorname")  
    private String firstName;  
  
    @Column(name = "Name")  
    private String lastName;  
  
    @Column(name = "Geburtstag")  
    private Date birthday;
```

```
@Entity  
@Table(name = "MitarbeiterJPA_Inheritance")  
public class Employee extends Person  
{  
    // KEINE ID, wird geerbt  
  
    @Column(name = "MitarbeiterNr")  
    private Integer employeeNumber;  
  
    @Column(name = "Abteilung")  
    private String workgroup;
```



---

# DEMO

**InheritanceWithJpaExample.java**

---

# Vererbungshierarchien – TABLE PER CLASS



## »Table per Class« – Eine Tabelle pro konkreter Klasse

Hierbei werden nur solche Klassen auf eine eigene Tabelle abgebildet, die tatsächlich instanziert werden können, die also nicht abstrakt sind. Im Gegensatz zur vorherigen Strategie werden abstrakte Basisklassen nicht als eigenständige Tabellen modelliert. Somit umfasst diese Strategie in der Regel jene Klassen ohne weitere Subklassen. Für jede konkrete Subklasse gibt es genau eine Tabelle, die die Attribute der korrespondierenden Klasse sowie all ihrer Basisklassen enthält: Durch diese Form der Speicherung befinden sich alle Daten einer Klasse in genau einem Datensatz einer Tabelle.

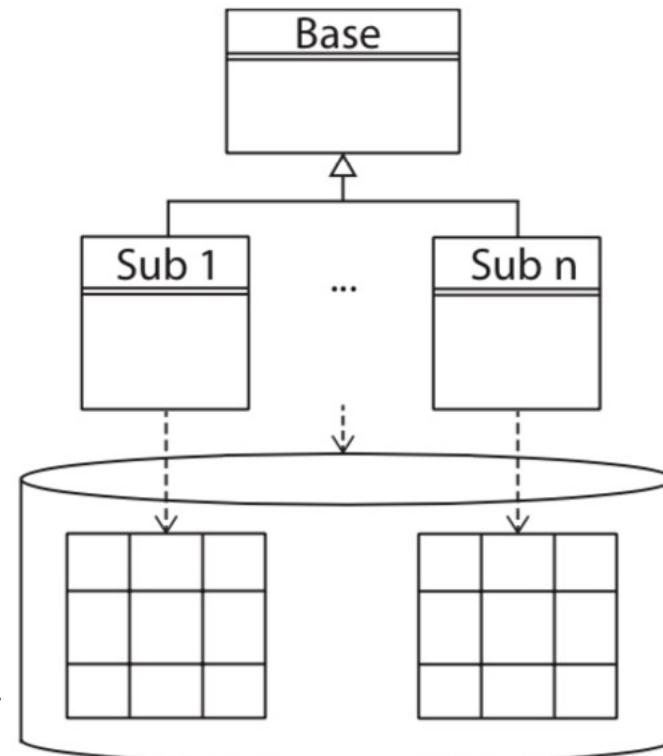
Beim Einsatz der Strategie »Table per Class« resultieren für die Vererbungshierarchie unseres Beispiels die zwei Tabellen Sportler und Mitarbeiter:

**Tabelle Sportler**

Name	Alter	HF-MIN	HF-MAX
Müller	27	48	198

**Tabelle Mitarbeiter**

Name	Alter	Abteilung	Titel
Meyer	33	Entwicklung	Dr.



# Vererbungshierarchien – TABLE PER CLASS

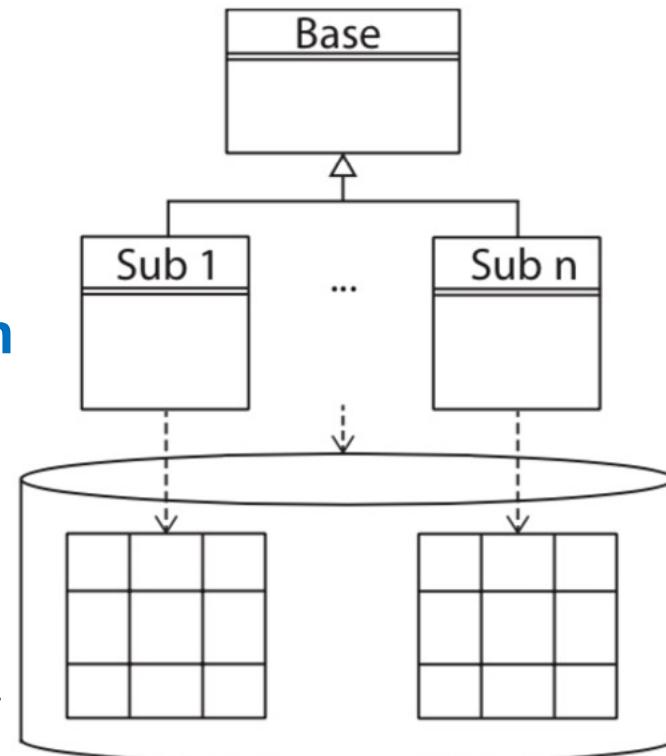


## Vorteile:

- Normalisiertes Schema, eine Datenbanktabelle für jede Klasse
- Not-Null-Constraints können definiert werden
- Fremdschlüsselreferenzen auf konkrete Unterklassen sind möglich (nicht aber auf abstrakte Basisklassen)

## Nachteile:

- Fast jeder Entity-Zugriff muss über mehrere Tabellen gehen
- Polymorphe Abfragen müssen auf mehrere Tabellen zugreifen



# Vererbungshierarchien mit Mapped Superclass

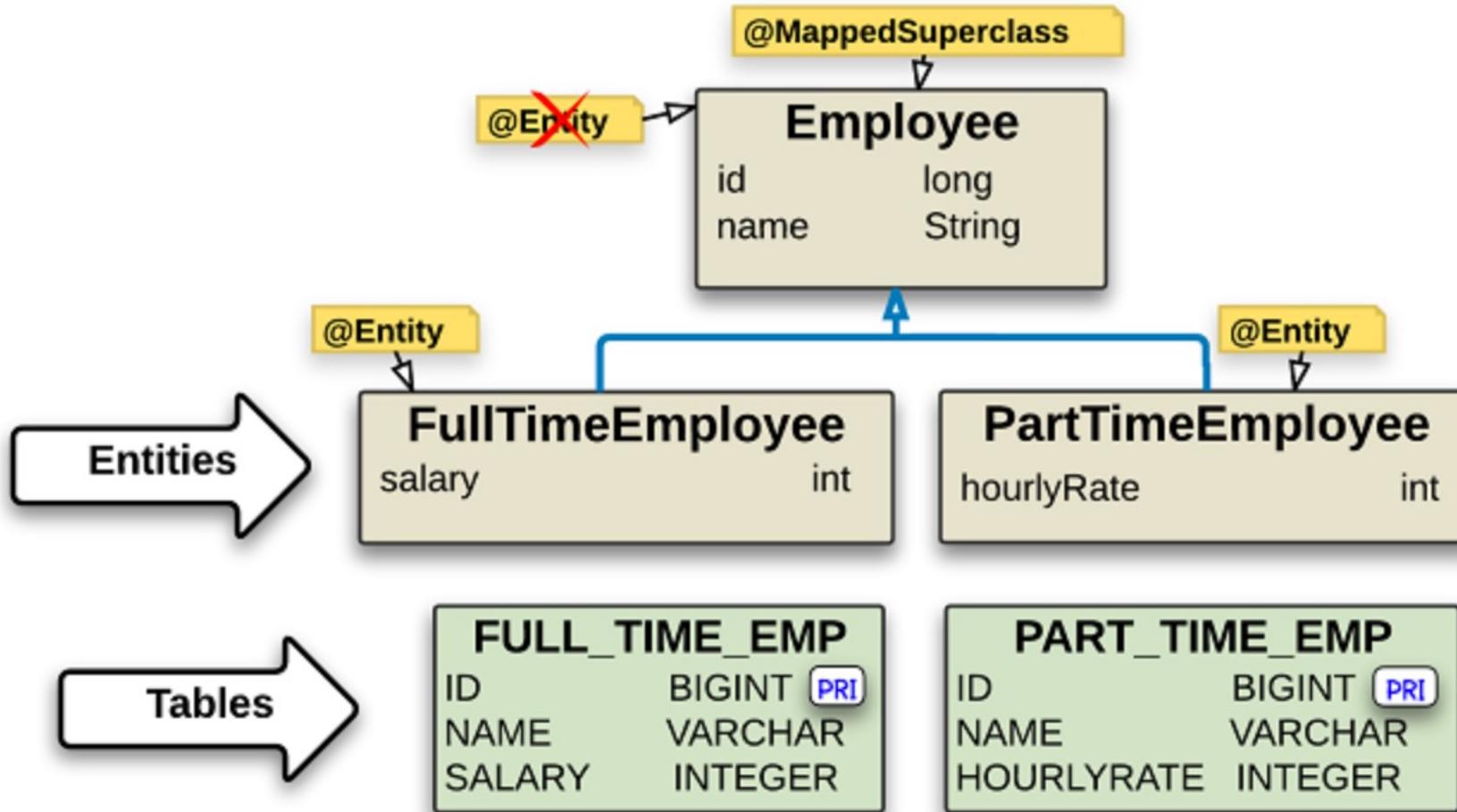
---



- Eine Mapped Superclass ermöglicht persistenten Zustand und Relationen, ist selbst aber keine Entity.
- Eine Mapped Superclass erlaubt im Gegensatz zu einer Entity keine Abfrage, Speicherung oder Beziehungen zur Superklasse.
- Die Annotation `@MappedSuperclass` wird verwendet, um eine Klasse zu kennzeichnen.
- Alle Mapping-Annotationen können genutzt werden, mit Ausnahme von `@Entity`.
- Es ist ähnlich wie die Table Per Class Strategie , aber im Datenmodell gibt es keine JOINS oder Vererbung. Es gibt keine Tabelle für die abgebildete Oberklasse.
- Vererbung existiert nur im Objektmodell.



## Mapped Superclasses



# Vererbungshierarchien



```
@MappedSuperclass  
public class Employee {  
    @Id  
    @GeneratedValue  
    private long id;  
    private String name;  
    .....  
}
```

```
@Entity  
@Table(name = "FULL_TIME_EMP")  
public class FullTimeEmployee extends Employee {  
    private int salary;  
    .....  
}
```

```
@Entity  
@Table(name = "PART_TIME_EMP")  
public class PartTimeEmployee extends Employee {  
    private int hourlyRate;  
    .....  
}
```

# Vererbung bei Entitäten

## (Abstract) Entitäten

- Basistyp für Entitäten ist abstrakte Entität
- Basistyp muss mit `@Entity` annotiert sein
- Vererbungsstrategie bestimmt Ablage des Basistyps
- Nach Entität-Basistypen kann abgefragt werden

## Mapped Supertypes

- Basistyp für Entitäten ist keine Entität, besitzt aber persistierbare Felder
- Basistyp muss mit `@MappedSuperclass` annotiert werden
- Mapped Supertypes haben keine eigenen Tabellen
- Mapped Supertypes können nicht abgefragt werden

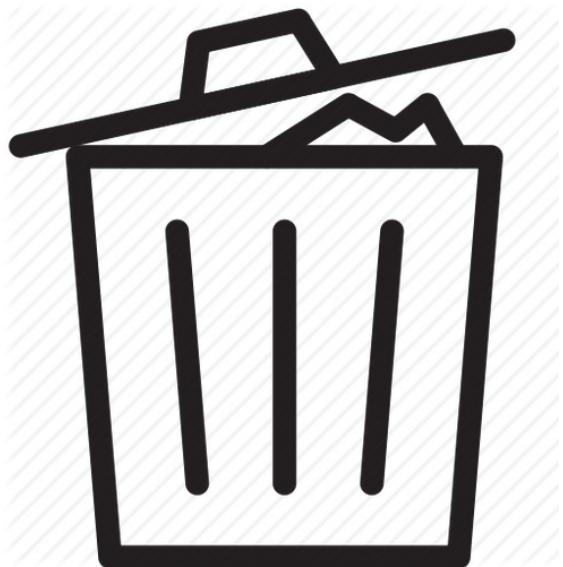
## Non-Entities

- Basistyp ist keine Entität und besitzt keine persistierbaren Felder
- Basistyp bei Persistierung komplett ignoriert



---

# Typische Probleme



## Typische Probleme

---

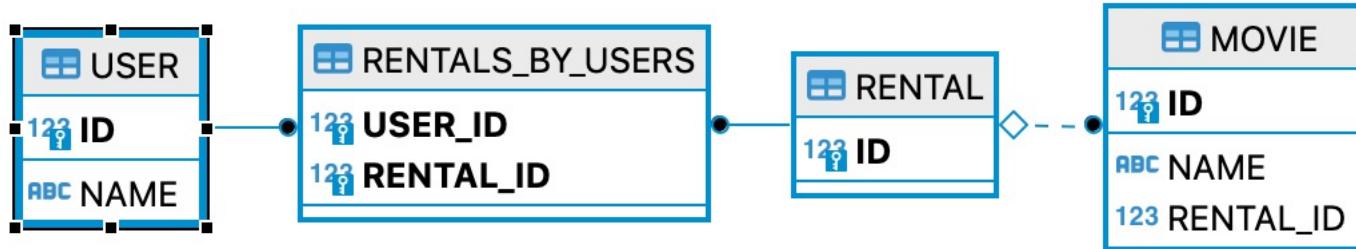


- Automatisches Laden (aller) assoziierten Objekte
  - FetchType.EAGER / LAZY
  - Entity Graph
- (Nicht)Löschen der assoziierten Objekte
  - Cascade.REMOVE / orphanRemoval oder nicht
- N + 1 Select Problem
  - JOIN FETCH (-> JPQL)

# N + 1 Select Probleme



- Assoziationen auf LAZY



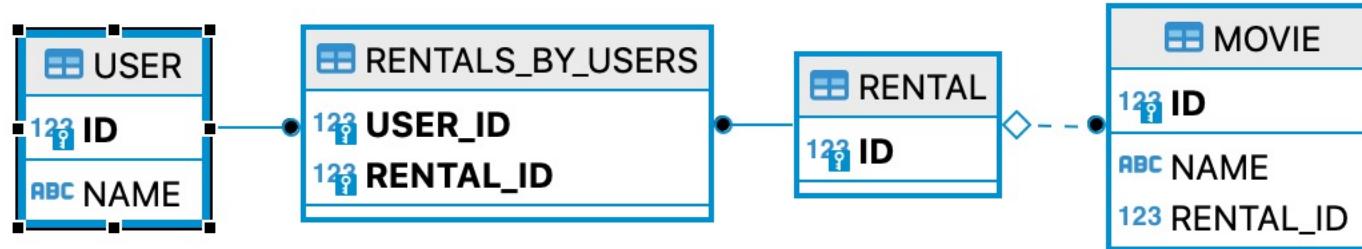
```
// N + 1
User michael = entityManager.find(User.class, 2L);
for (Rental rental : michael.getRentals())
{
    System.out.println(rental);
}
```

1. User wird geladen
2. alle assoziierten Rentals werden geladen

# N + 1 Select Probleme



- **Assoziationen auf LAZY**



```
// N + 1
User userPeter = entityManager.find(User.class, 1L);
for (Rental rental : userPeter.getRentals())
{
    for (Movie movie : rental.getMovies())
        System.out.println(movie);
}
```

1. User wird geladen
2. alle assoziierten Rentals werden geladen
3. alle ausgeliehenen Filme werden geladen



---

# Entity Graph



## Entity Graph

---



- **Bisher: Relationen lassen sich mit FetchType entweder als EAGER oder LAZY kennzeichnen**
  - **EAGER lädt oftmals zu viele Daten, LAZY oftmals besser, führt aber zum Nachladen von Daten**
  - **EAGER und LAZY sind aber zur Compile-Zeit fix**
  - **Praktisch wäre es, für einzelne Queries, die Vorgaben ändern zu können**
  - **Hier kommen Entity Graphs ins Spiel: Diese erlauben es, die einzulesenden Assoziationen dynamisch zu bestimmen.**
-



- **Entitätsgraphen erlauben unabhängig von der Abfrage zu definieren, welche Attribute aus der Datenbank geholt werden sollen.**
- **Ein Entity-Graph kann als Fetch- oder als Load-Graph verwendet werden.**
  - Beim Fetch-Graph werden nur die vom Entity-Graph angegebenen Attribute als **EAGER** behandelt, **alle anderen Attribute als "lazy"**.
  - Beim **Load-Graph** wie vom Entity-Graph vorgeben, alle anderen Attribute, die nicht angegebenen, behalten ihren **Standard-Fetch-Typ**.
- <https://blog.doubleslash.de/entitygraphs-dynamischer-performanceschub-bei-datenbankabfragen-mit-jpa/>

# Entity Graph



- Rot: Es wurden neben den geforderten auch unnötige Daten mitgeladen
- Orange: Es mussten Felder nachgeladen werden
- Grün: Es wurden exakt die Daten geladen, die auch benötigt werden

Geforderte Attribute	Ohne Optimierung	Fetchtype (Addresses, Items = LAZY)	EntityGraph
Customer	Customer, Addresses, Bills, Items	Customer, Bills	Customer
Customer, Addresses	Customer, Addresses, Bills, Items	Customer, Addresses Bills	Customer, Addresses
Customer, Bills	Customer, Addresses, Bills, Items	Customer, Bills	Customer, Bills
Customer, Addresses, Bills	Customer, Addresses, Bills, Items	Customer, Addresses, Bills	Customer, Addresses, Bills
Customer, Addresses, Bills, Items	Customer, Addresses, Bills, Items	Customer, Addresses, Bills, Items	Customer, Addresses, Bills, Items

## Entity Graph -- Ausgangsbasis

---



```
@Entity(name = "BLOG_POST")
public class Post
{
    @Id
    @GeneratedValue
    private Long id;
    private String subject;

    private String additionalInfo;
    private LocalDate timeStamp;

    @OneToMany(mappedBy = "post", fetch = FetchType.LAZY)
    private List<Comment> comments = new ArrayList<>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn
    private Member member;
    //...
}
```

# Entity Graph – Zwei Entity Graphs

---



```
@NamedEntityGraph(  
    name = "post-entity-graph-plain-subject",  
    attributeNodes = {  
        @NamedAttributeNode("subject"),  
    }  
)  
@NamedEntityGraph(  
    name = "post-entity-graph",  
    attributeNodes = {  
        @NamedAttributeNode("subject"),  
        @NamedAttributeNode("member"),  
        @NamedAttributeNode("comments"),  
    }  
)  
  
@Entity(name = "BLOG_POST")  
public class Post
```

## Entity Graph bei Abfrage nutzen / dynamisch definieren



- **getEntityGraph() liefert Zugriff auf vordefinierte Entity Graphs**

```
EntityGraph entityGraph = entityManager.getEntityGraph("post-entity-graph");
```

```
Map<String, Object> properties = new HashMap<>();  
properties.put("javax.persistence.fetchgraph", entityGraph);
```

```
Post postEG = entityManager.find(Post.class, 1L, properties);
```

- **createEntityGraph() ermöglicht dynamische Definition von Entity Graphs**

```
EntityGraph dynamicEntityGraph = entityManager.createEntityGraph(Post.class);  
dynamicEntityGraph.addAttributeNodes("subject");  
dynamicEntityGraph.addAttributeNodes("member");
```



---

# DEMO

`EntityGraphExample.java`

---



---

## Exercises Part 3 & 4

<https://github.com/Michaeli71/JPA-Workshop.git>





# PART 5: JPQL

## Warum JPQL?

---



- **EntityManager.find() liefert nur einzelne Entities und nur über Primärschlüssel**
  - **Für komplexere Abfragen: JPQL (Java Persistence Query Language)**
  - **JPQL ist eine deklarative, SQL-ähnliche Abfragesprache**
  - **Wird zur Laufzeit in natives SQL übersetzt**
  - **Auch natives SQL möglich, falls JPQL nicht ausreicht**
-



- JPQL kann mit verschiedenen Datenbanken verwendet werden, wie z. B. HSQLDB, MySQL, Oracle, Postgres
- JPQL arbeitet auf Java-Objekten (nicht auf Tabellen)
- JPQL-Abfragen liefern Objekte oder Collections (nicht ResultSets)
- JPQL ist einfach und robust, aber mächtig
  - JPQL kann Join-Operationen durchführen, aber in Form von ..-Notation:  
`rental.movie.priceCategory`
  - JPQL kann Daten in einem Batch aktualisieren und löschen.
  - JPQL kann Aggregatfunktionen mit Sortierung und Gruppierungsklauseln ausführen



## JPQL ist eine mächtige Sprache

ABS	CONCAT	GROUP	MOD	SUM
ALL	COUNT	HAVING	NEW	THEN
AND	CURRENT_DATE	IN	NOT	TRAILING
ANY	CURRENT_	INDEX	NULL	TRIM
AS	TIMESTAMP	INNER	NULIF	TRUE
ASC	DELETE	IS	OBJECT	TYPE
AVG	DESC	JOIN	OF	UNKNOWN
BETWEEN	DISTINCT	KEY	OR	UPDATE
BIT_LENGTH	ELSE	LEADING	ORDER	UPPER
BOTH	EMPTY	LEFT	OUTER	VALUE
BY	END	LENGTH	POSITION	WHEN
CASE	ENTRY	LIKE	SELECT	WHERE
CHAR_LENGTH	ESCAPE	LOCATE	SET	
CHARACTER_	EXISTS	LOWER	SIZE	
LENGTH	FALSE	MAX	SOME	
CLASS	FETCH	MEMBER	SQRT	
COALESCE	FROM	MIN	SUBSTRING	



- JPQL unterstützt
  - SELECT, UPDATE, DELETE
- **Queries repräsentiert durch javax.persistence.TypedQuery, erzeugt über Factory-Methoden im EntityManager**
- **Können zu verschiedenen Zeiten definiert werden:**
  - Dynamische Queries zur Laufzeit
  - Statische (named) Queries zur Compilezeit => Vordefinierte Queries an Entity-Klassen
- **Kommandos in zwei Sprachen möglich:**
  - Java Persistence Query Language (JPQL)
  - Standard Query Language (SQL)



# Queries



# JPQL – Ausgangsbasis Entity und Tabelle



```
@Entity  
@Table(name = "PersonenJPQL")  
public class Person implements Serializable  
{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    @Column(name = "Vorname")  
    private String firstName;  
  
    @Column(name = "Name")  
    private String lastName;  
  
    @Column(name = "Geburtstag")  
    private Date birthday;  
  
    @Column(name = "Alter")  
    private int age;  
  
    ...
```

PERSONENJPQL	
123	ID
123	ALTER
⌚	GEBURTSTAG
ABC	VORNAME
ABC	NAME

# Queries



- **Query createQuery(String jpqlString)**

```
Query query = entityManager.createQuery("SELECT p FROM Person p");
List resultList = query.getResultList();
resultList.forEach(System.out::println);
```

OBJEKT  
MODELL

- **Query createNativeQuery(String sqlString)**

```
final String sqlString = "SELECT VORNAME, NAME FROM PersonenJPQL";
final Query nativeQuery = entityManager.createNativeQuery(sqlString);
```

DB  
MODELL

```
System.out.println("\nNative Query");
final List<Object[]> rows = nativeQuery.getResultList();
rows.forEach(row -> System.out.println(Arrays.toString(row)));
```

## Queries

---



- **Query createNamedQuery(String queryName)**

```
private static void performNamedQuery(final EntityManager entityManager)
{
    final Query namedQuery = entityManager.createNamedQuery("findPersonsBornAfter");
    namedQuery.setParameter("birthday", LocalDate.of(1977, 1, 1));

    final List<Person> result = namedQuery.getResultList();
    result.forEach(System.out::println);
}
```

```
@Entity
@Table(name = "PersonenJPQL")
@NamedQuery(query = "SELECT p FROM Person p WHERE p.birthday > :birthday",
            name = "findPersonsBornAfter")
public class Person implements Serializable
```

---



- **Query `createNamedQuery(String jpqlString)`**

```
String personsQuery = "SELECT p FROM Person p";
TypedQuery<Person> persons = entityManager.createQuery(personsQuery,
                                                       Person.class);
List<Person> allPersons = persons.getResultList();

// Zugriffe auf einzelne Attribute
String firstNamesQuery = "SELECT p.firstName FROM Person p " +
                         "ORDER BY p.firstName";
TypedQuery<String> firstNames = entityManager.createQuery(firstNamesQuery,
                                                          String.class);
List<String> sortedFirstNames = firstNames.getResultList();
```

# Aggregationen in Abfragen

---



- Wie in SQL kann man auch in JPQL verschiedene vordefinierte Funktionen innerhalb von Abfragen nutzen. Damit lassen sich etwa Statistiken berechnen oder Texte manipulieren. Hier eine Auswahl gebräuchlicher Funktionen:
  - COUNT – Zählt die Anzahl an Einträgen einer Abfrage.
  - MAX / MIN – Ermittelt das Maximum / Minimum der Werte einer Spalte.
  - AVG – Berechnet den Durchschnitt der Werte einer Spalte.
  - SUM – Summiert die Werte einer Spalte.
  - TRIM – Entfernt Whitespaces am Anfang und Ende eines Strings.
  - LENGTH – Ermittelt die Länge eines Strings.
  - UPPER / LOWER – Wandelt in Groß-/Kleinschreibung um.



```
// Scalar function
Query query1 = entityManager.createQuery("SELECT UPPER(p.lastName) from Person p");
List<String> list = query1.getResultList();
list.forEach(lastName -> System.out.println("NAME: " + lastName));

// Aggregate function
Query query2 = entityManager.createQuery("SELECT MAX(p.age) from Person p");
Integer result = (Integer)query1.getSingleResult();
System.out.println("Max Age :" + result);

// Berechnungen in Queries
String personCountQuery = "SELECT COUNT(p) FROM Person p";
Query countQuery = entityManager.createQuery(personCountQuery);
Object count = countQuery.getSingleResult();

String avgAgeQuery = "SELECT AVG(p.age) FROM Person p";
TypedQuery<Double> avgQuery = entityManager.createQuery(avgAgeQuery, Double.class);
Double avgAge = avgQuery.getSingleResult();
```

# Parametrierte Abfragen



```
final String jpql1 = "SELECT person FROM Person person" +  
    " WHERE person.age >= " + minAge +  
    " AND person.age < " + maxAge;
```



```
String jpql1 = "SELECT person FROM Person person " +  
    "WHERE person.age >= ? AND person.age < ?";  
TypedQuery<Person> typedQuery1 = entityManager.createQuery(jpql1, Person.class);  
typedQuery1.setParameter(1, minAge);  
typedQuery1.setParameter(2, maxAge);  
typedQuery1.getResultList().forEach(System.out::println);
```

```
String jpql2 = "SELECT person FROM Person person " +  
    "WHERE person.age >= :minAge AND person.age < :maxAge";  
TypedQuery<Person> typedQuery2 = entityManager.createQuery(jpql2, Person.class);  
typedQuery2.setParameter("minAge", minAge);  
typedQuery2.setParameter("maxAge", maxAge);  
typedQuery2.getResultList().forEach(System.out::println);
```



---

# DEMO

**PersonJpqlQueriesExample.java**  
**ScalarAndAggregateFunctionsExample.java**

# Query mit Objekterzeugung



- **Mitunter möchte man in einer Query nur eine Teilmenge der Attribute, etwa Name und Alter einer Person, auslesen.**
- **Man kann daraus direkt spezialisierte Datenbehälterobjekte erzeugen.**

```
String createObjJPQL = "SELECT NEW part5_jpql.intro.NameAndAge"  
                      + "(p.firstName, p.age) FROM Person p";  
TypedQuery<NameAndAge> query = entityManager.createQuery(createObjJPQL,  
NameAndAge.class);
```

```
public class NameAndAge  
{  
    private final String name;  
    private final int age;  
  
    public NameAndAge(final String name, final int age)  
    {  
        this.name = name;  
        this.age = age;  
    }  
    // ....  
}
```

# JPQL Pagination

---



// Einfache Abfrage

```
String personsQuery = "SELECT p FROM Person p";
TypedQuery<Person> personsQuery = entityManager.createQuery(personsQuery,
                                         Person.class);
personsQuery.setFirstResult(5);
personsQuery.setMaxResults(10);
List<Person> allPersons = personsQuery.getResultList();
```

// Zugriffe auf einzelne Attribute

```
String firstNamesQuery = "SELECT p.firstName FROM Person p " + "ORDER BY p.age";
TypedQuery<String> firstNamesQuery = entityManager.createQuery(firstNamesQuery,
                                         String.class);
firstNamesQuery.setFirstResult(15);
firstNamesQuery.setMaxResults(20);
List<String> sortedFirstNames = firstNamesQuery.getResultList();
```

---

## JPQL – Update & Delete

---



```
private static void performUpdateAndDelete(final EntityManager entityManager)
{
    final String updateJPQL = "UPDATE Person p SET p.firstName = 'Mike' " +
        "WHERE p.firstName LIKE 'Micha%'";
    final Query updateCommand = entityManager.createQuery(updateJPQL);
    final int updatedRows = updateCommand.executeUpdate();

    final String deleteJPQL = "DELETE FROM Person p WHERE p.firstName LIKE 'T%'";
    final Query deleteCommand = entityManager.createQuery(deleteJPQL);
    final int deletedRows = deleteCommand.executeUpdate();

    System.out.println("\nUpdated rows: " + updatedRows);
    System.out.println("Deleted rows: " + deletedRows);
}
```



---

# DEMO

`PagedJpqlQueriesExample.java`  
`PersonUpdateAndDeleteExample.java`

---



---

# Queries – JOIN FETCH



## Typische Probleme

---



- Bei JPQL-Queries werden normalerweise keine abhängigen Entities mitgeladen
- Als Folge: Wenn Entity später detached wird, dann sind assoziierte Elemente leer (sofern nicht zwischenzeitlich nachgeladen)
- Als Abhilfe: JOIN FETCH
- Beispiel

```
SELECT DISTINCT t FROM Team t JOIN FETCH t.teamMembers  
WHERE t.team.name = :teamName
```

## N + 1 Select Probleme

---



```
System.out.println("-- executing query N + 1 --");

// N + 1
User michael = entityManager.find(User.class, 2L);
for (Rental rental : michael.getRentals())
    System.out.println(michael.getName() + " - " + rental);

entityManager.clear();

System.out.println("-- executing query N + 1 ^ 2--");

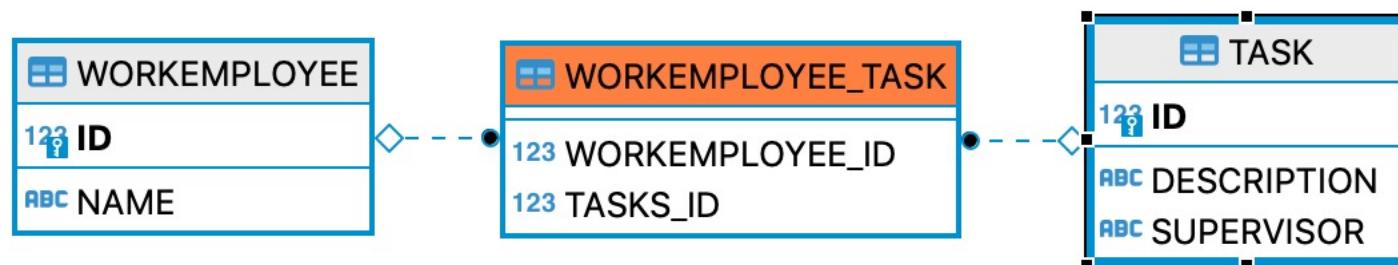
// N + 1 "im Quadrat"
User userPeter = entityManager.find(User.class, 1L);
for (Rental rental : userPeter.getRentals())
{
    for (Movie movie : rental.getMovies())
        System.out.println(movie);
}
```

# N + 1 Select Probleme -- Datenmodell



```
@Entity  
public class WorkEmployee  
{  
    @Id  
    @GeneratedValue  
    private long id;  
    private String name;  
    @ManyToMany(cascade = CascadeType.ALL,  
               fetch = FetchType.LAZY)  
    private List<Task> tasks = new ArrayList<>();
```

```
@Entity  
public class Task  
{  
    @Id  
    @GeneratedValue  
    private long id;  
    private String description;  
    private String supervisor;
```



## N + 1 Select Probleme



```
Query query = entityManager.createQuery("SELECT DISTINCT e FROM WorkEmployee e  
INNER JOIN e.tasks t");  
List<WorkEmployee> resultList = query.getResultList();  
  
for (WorkEmployee employee : resultList) {  
    System.out.println(employee.getName() + " - " + employee.getTasks());  
}
```

Hibernate:

```
select  
    distinct workemploy0_.id as id1_2_,  
    workemploy0_.name as name2_2_  
from  
    WorkEmployee workemploy0_  
inner join  
    WorkEmployee_Task tasks1_  
        on  
    workemploy0_.id=tasks1_.WorkEmployee_id  
inner join  
    Task task2_  
        on tasks1_.tasks_id=task2_.id
```

## N + 1 Select Probleme



```
for (WorkEmployee employee : resultList) {  
    System.out.println(employee.getName() + " - " + employee.getTasks());  
}
```

Hibernate:

```
select  
    tasks0_.WorkEmployee_id as workempl1_3_0_,  
    tasks0_.tasks_id as tasks_id2_3_0_,  
    task1_.id as id1_1_1_,  
    task1_.description as descript2_1_1_,  
    task1_.supervisor as supervis3_1_1_  
from  
    WorkEmployee_Task tasks0_  
inner join  
    Task task1_  
        on tasks0_.tasks_id=task1_.id  
where  
    tasks0_.WorkEmployee_id=?  
Diana - [Task [id=2, description=Coding, supervisor=Denise],  
Task [id=3, description=Designing, supervisor=Denise]]
```

# N + 1 Select Probleme – Lösung JOIN FETCH



```
SELECT DISTINCT e FROM WorkEmployee e INNER JOIN FETCH e.tasks t"
```

Hibernate:

```
select
    distinct workemploy0_.id as id1_2_0_,
    task2_.id as id1_1_1_,
    workemploy0_.name as name2_2_0_,
    task2_.description as descript2_1_1_,
    task2_.supervisor as supervis3_1_1_,
    tasks1_.WorkEmployee_id as workempl1_3_0__,
    tasks1_.tasks_id as tasks_id2_3_0__
from
    WorkEmployee workemploy0_
inner join
    WorkEmployee_Task tasks1_
        on workemploy0_.id=tasks1_.WorkEmployee_id
inner join
    Task task2_
        on tasks1_.tasks_id=task2_.id
```

Diana - [Task [id=2, description=Coding, supervisor=Denise], Task [id=3, description=Designing, supervisor=Denise]]  
Mike - [Task [id=5, description=Refactoring, supervisor=Rose], Task [id=6, description=Documentation, supervisor=Mike]]



# DEMO

**JoinFetchExample1-3.java**

---



---

# Queries & Criteria API



## Vergleich mit Criteria API



```
private static void calcAvgAge(final EntityManager entityManager)
{
    String avgAgeJPQL = "SELECT AVG(person.age) FROM Person person";
    TypedQuery<Double> query1 = entityManager.createQuery(avgAgeJPQL, Double.class);

    CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
    CriteriaQuery<Double> criteriaQuery = criteriaBuilder.createQuery(Double.class);
    Root<Person> person = criteriaQuery.from(Person.class);
    criteriaQuery.select(criteriaBuilder.avg(person.get("age")));
    TypedQuery<Double> query2 = entityManager.createQuery(criteriaQuery);

    final Double avgAge1 = query1.getSingleResult();
    final Double avgAge2 = query2.getSingleResult();
    System.out.println("\nAvg Age 1: " + avgAge1 + " / Age 2: " + avgAge2);
}
```

## Vergleich mit Criteria API



```
private static void personsWithinAgeRange(final EntityManager entityManager,
                                         final int minAge, final int maxAge)
{
    String selectJPQL = "SELECT person FROM Person person "
                        "WHERE person.age >= :minAge AND person.age < :maxAge";
    TypedQuery<Person> typedQuery1 = entityManager.createQuery(selectJPQL, Person.class);
    typedQuery1.setParameter("minAge", minAge);
    typedQuery1.setParameter("maxAge", maxAge);

    CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
    CriteriaQuery<Person> criteriaQuery = criteriaBuilder.createQuery(Person.class);
    Root<Person> person = criteriaQuery.from(Person.class);
    Path<Integer> age = person.get("age");
    criteriaQuery.where(criteriaBuilder.and(criteriaBuilder.greaterThanOrEqualTo(age, minAge),
                                             criteriaBuilder.lessThan(age, maxAge)));
    TypedQuery<Person> typedQuery2 = entityManager.createQuery(criteriaQuery);

    final List<Person> results1 = typedQuery1.getResultList();
    final List<Person> results2 = typedQuery2.getResultList();
    System.out.println("Results1: " + results1);
    System.out.println("Results2: " + results2);
}
```



# DEMO

**PersonJpqlAndCriteriaApiExample.java**

---

## Fazit JPQL vs Criteria API

---



- **JPQL basiert auf Strings**
  - **JPQL nicht zwar typsicher (-)**
  - **Nicht Refactoring-sicher (-)**
  - **aber sehr einfach in der Handhabung (++)**
  - **Große Analogie zu SQL hilft beim Einstieg und er Verständlichkeit (++)**
  
  - **Criteria-API basiert auf Java-Objekten**
  - **Criteria-API ist typsicher, Compiler prüft auf Gültigkeit (+)**
  - **Refactoring-sicher (+)**
  - **Deutlich komplexer in der Aufbereitung (--)**
  - **Schwierig zu lesen und nachzuvollziehen (--)**
-



---

# Queries & Indexing



# Indexing

---



- JPA bietet erst seit Version 2.1 mit **@Index** vernünftigen Support  
**(ACHTUNG: Nur wenn JPA Tables generiert!)**
  - Abfragen ohne Index erfordern lineare Suche über alle Datensätze einer Tabelle => **SLOW**
  - Standardmäßig existiert immer ein Index über den Primärschlüssel => **FAST**
  - Oftmals werden weitere Indexe über andere Spalten benötigt, um Abfragen zu beschleunigen
  - Index kostet aber Speicherplatz und Verwaltungsoverhead (muss aktuell gehalten werden)
-

# Indexing

---



- **Index mit @Index definieren**

```
@Table(indexes = @Index(columnList = "firstName"))
```

```
@Index(name = "firstname_index", columnList = "firstName")
```

- **Mehrere Spalten**

```
@Index(name = "mulitIndex1", columnList = "firstName, lastName")
```

```
@Index(name = "mulitIndex2", columnList = "lastName, firstName, birthday DESC")
```

- **Eindeutigkeit**

```
@Index(name = "uniqueMulitIndex", columnList = "firstName, lastName, birthday", unique = true)
```

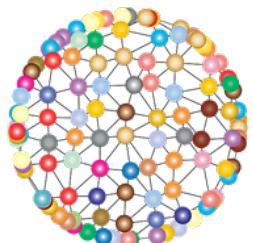
---

# Indexing



- **Multiple Indexes**

```
@Entity  
@Table(indexes = {  
    @Index(columnList = "firstName"),  
    @Index(name = "fn_index", columnList = "firstName"),  
    @Index(name = "mulitIndex1", columnList = "firstName, lastName"),  
    @Index(name = "mulitIndex2", columnList = "lastName, firstName"),  
    @Index(name = "mulitSortIndex", columnList = "firstName, lastName DESC"),  
    @Index(name = "uniqueIndex", columnList = "firstName", unique = true),  
    @Index(name = "uniqueMulitIndex", columnList = "firstName, lastName", unique = true)  
})  
public class Student implements Serializable
```



**Welche Indexe  
brauchen wir?**

# Indexing

---



- **Große Faustregel:**
  - Alle Queries analysieren (betrachten)
  - Für alle Spalten, die in WHERE auftauchen einen Index erstellen
  - Reihenfolge beachten
  - Sortierung beachten
- **Defining Indexes**

<https://dzone.com/articles/8-rules-to-create-useful-database-indexes>

---



---

## Exercises Part 5

<https://github.com/Michaeli71/JPA-Workshop.git>





# PART 6: Transaktionen

# Transaktionen Grundlagen

---



- Die in einer Datenbank gespeicherten Daten werden oftmals von verschiedenen Anwendungen oder von mehreren Threads der eigenen Anwendung parallel genutzt.
- **Transaktion** = Abfolge von Anweisungen, die im Resultat entweder vollständig als eine atomare Gesamtaktion oder überhaupt nicht ausgeführt wird.
- **Commit** – Können alle Verarbeitungsschritte oder Anweisungen einer Transaktion erfolgreich bearbeitet werden, so wird die Transaktion **als Ganzes erfolgreich beendet** und das Ergebnis in der Datenbank gesichert. Man spricht dann von einem **Commit**.
- **Rollback** – Tritt dagegen bei einem der Verarbeitungsschritte ein **Fehler** auf, so sollten alle zuvor gemachten Änderungen automatisch von der Datenbank rückgängig gemacht werden. Durch **Rollback** befindet sich die Datenbank anschließend für die betroffenen Datensätze wieder in einem Zustand, der dem entspricht, wie er vor der Transaktion war.

# Transaktionen Eigenschaften

---



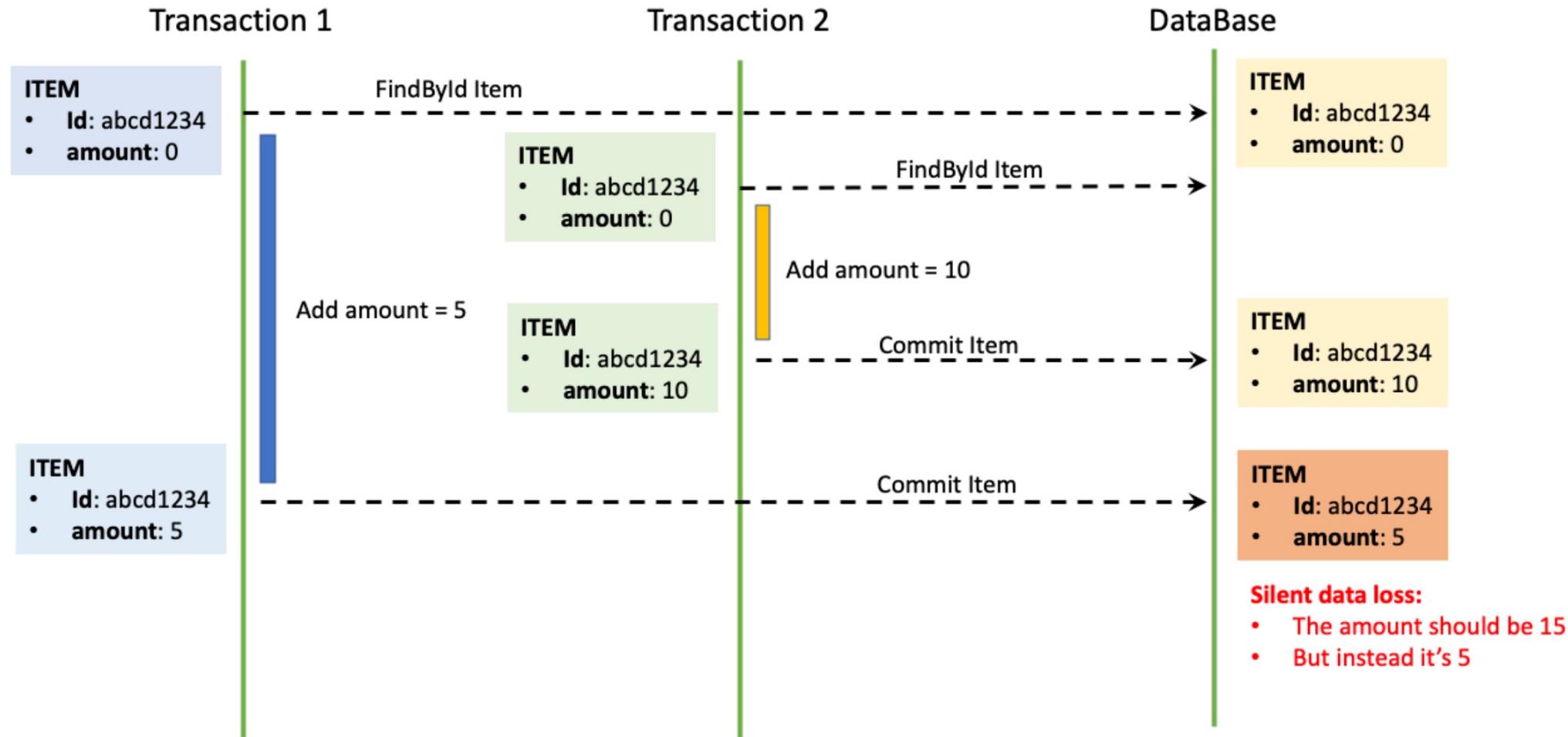
- **A – Atomicity** – Die gesamte Transaktion erfolgt (logisch) atomar, d. h., sie wirkt, wie eine unteilbare Gesamtaktion, obwohl sie aus Teilaktionen besteht.
- **C – Consistency** – Die Aktionen einer Transaktion hinterlassen die Datenbank in einem konsistenten Zustand, sofern sich die Datenbank zuvor in einem solchen befand.
- **I – Isolation** – Eine Transaktion wird (inhaltlich) *nicht* durch andere Aktionen, etwa parallel stattfindende Zugriffe und Transaktionen, beeinflusst.\*
- **D – Durability** – Die Änderungen werden dauerhaft gespeichert.

\*Um trotz konkurrierender Zugriffe einen guten Durchsatz sowie eine hohe Performance zu erzielen, erlauben Datenbanksysteme verschiedene Lockerungen bezüglich der Forderung nach Isolation.

# Transaktionen Probleme bei fehlender Isolation / Lost-Update



- **Problem:** Es ist potenziell möglich, dass Datensätze konkurrierend verändert werden.





- Nur drei Schritte doch schwerwiegende Probleme:
  1. Transaktion A lädt Daten.
  2. Transaktion B lädt und verändert die Daten und führt einen Commit aus.
  3. Basierend auf den im ersten Schritt gelesenen (und mittlerweile veralteten) Daten setzt Transaktion A die Arbeit fort und schreibt Ergebnisse in die Datenbank.
- Konkurrierende Zugriffe problematisch. Als Abhilfe keine Unterbrechung zulassen
- ABER: alle Tabellen, die in einer Transaktion angesprochen werden, während der Zeitdauer dieser Transaktion nicht durch andere Transaktionen abgefragt oder modifiziert werden können. => andere Transaktionen werden aufgehalten.
- **Nicht nötig, etwa wenn die Transaktionen auf unterschiedlichen Datensätzen arbeiten.**
- **Deshalb existieren die verschiedene Isolationslevel.**



**Lockungen bezüglich der Isolation von Transaktionen, um den Grad an Parallelität zu erhöhen:**

- **TRANSACTION\_READ\_UNCOMMITTED** – Auch solche Werte/Zwischenzustände sichtbar, die durch Aktionen in anderen Transaktionen verändert, aber noch nicht committet sind. Nicht garantiert, dass diese Werte jemals permanent in der Datenbank gespeichert werden. *Dirty Reads*
- **TRANSACTION\_READ\_COMMITTED** – Nur solche Werte zugreifbar, die bereits committet wurden. Allerdings können andere Transaktionen noch während der Verarbeitung der Daten Änderungen am Datenbankinhalt vornehmen. Wenn die Abfragen wiederholt werden, ergeben sich dadurch dann andere Wertbelegungen oder sogar andere Datensätze als Ergebnis. *Non Repeatable Reads*

# Transaktionen Isolationslevel



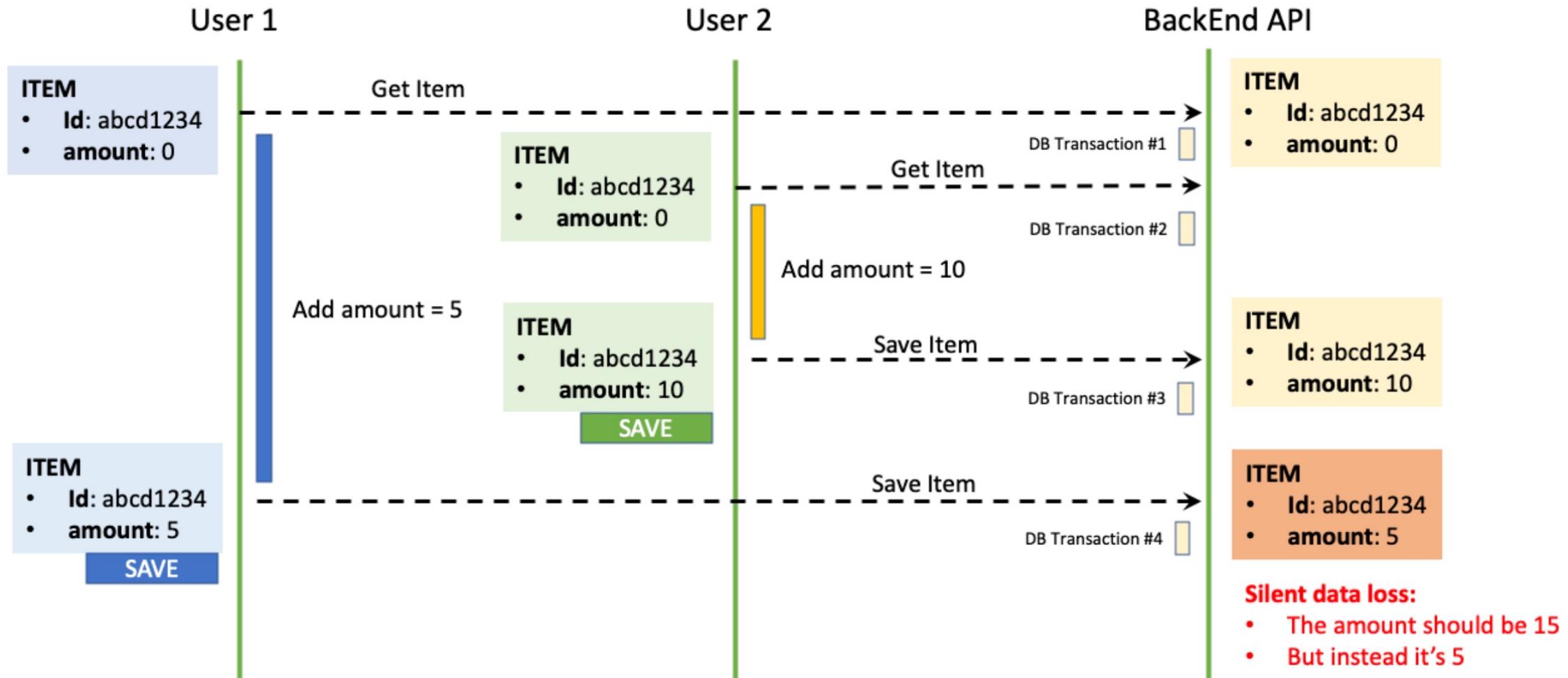
- **TRANSACTION\_REPEATABLE\_READ** – Nur committe Daten lesbar und es spiegeln sich keine Änderungen anderer Transaktionen in den aktuellen Daten wider. Allerdings könnten Datensätze hinzugefügt oder gelöscht worden sein. **Phantom Reads**
- **TRANSACTION\_SERIALIZABLE** – Transaktionen werden so ausgeführt, als ob sie tatsächlich unabhängige Aktionen wären. Vollständig voneinander abgeschottet: keine Aktion (z. B. Einfügen, Ändern, Löschen) ist für eine andere Transaktion sichtbar.

Isolationsebene	Dirty Read	Lost Update	Non-Repeatable Read	Phantom
<i>Read Uncommitted</i>	möglich	möglich	möglich	möglich
<i>Read Committed</i>	unmöglich	möglich	möglich	möglich
<i>Repeatable Read</i>	unmöglich	unmöglich	unmöglich	möglich
<i>Serializable</i>	unmöglich	unmöglich	unmöglich	unmöglich

# Transaktionen Probleme bei Langläufern



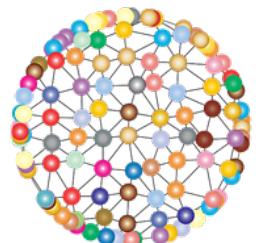
- **Problem:** Lange Laufzeiten mehrere Transaktionen





---

**Was können wir tun, um  
Probleme erkennen zu  
können?**



## Locking Strategien

---



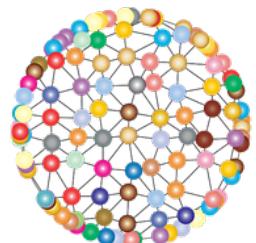
- JPA 2 unterstützt sowohl Optimistic Locking als auch Pessimistic Locking.
- Das Sperren ist wichtig, um Dateninkonsistenzen zu vermeiden, die durch Aktualisierungen derselben Daten etwa durch parallel arbeitende Benutzer entstehen.
- **Optimistic Locking** basiert auf dem **Erkennen von Änderungen** und wird beim Transaktions-Commit angewendet. Jedes zu löschen / aktualisierte Datenbankobjekt wird überprüft. Eine Exception wird ausgelöst, wenn eine bereits durch eine andere Transaktion vorgenommene Aktualisierung festgestellt wird.
- **Pessimistic Locking** ermöglicht es, gewisse Entities mithilfe von Locks zu sperren (Schreib-Lese-Sperre / Nur-Schreib-Sperre)



---

# Optimistic Locking





**Aber: Wie erkennt man  
nun Modifikationen?**

## Optimistic Locking

---



- Recht unpraktisch wäre es, die Daten an sich vergleichen zu müssen.
- Es gibt aber eine praktikablere Lösung: Jedes Objekt besitzt eine Versionsnummer, die bei Modifikationen hochgezählt wird. Auf diese Weise lassen sich beim **Optimistic Locking** konkurrierende Schreibzugriffe durch Abweichungen in der Versionsnummer erkennen.
- Je nach Datenbank werden die Versionsnummern intern verwaltet, können aber durch die Definition eines Versionsfelds sichtbar/zugreifbar gemacht werden.

# Optimistic Locking

---



- Vor den Aktionen auf einem Objekt wird die gerade aktuelle Versionsnummer ausgelesen und zwischengespeichert. Dann erfolgen die Aktionen.
  - Beim späteren Schreibvorgang wird geprüft, ob der derzeit in der Datenbank gespeicherte Datensatz immer noch die zuvor gemerkte Versionsnummer besitzt.
  - Stimmt die Version noch überein, hat zwischenzeitlich keine Änderung stattgefunden. Das Schreiben ist problemlos möglich.
  - Wird eine andere Versionsnummer vorgefunden, so muss die Transaktion abgebrochen werden, weil ein konkurrierender Schreibzugriff vorliegt.
- 
- **Beispiel in SQL**

```
UPDATE Personen SET Name = 'Changed', VERSION = 7  
WHERE ID = 4711 AND VERSION = 6;
```

# Optii

## Transaction 1

**ITEM**  
• Id: abcd1234  
• amount: 0  
• version: 0

FindById Item

Add amount = 5

**ITEM**  
• Id: abcd1234  
• amount: 5  
• version: 0

**ITEM**  
• Id: abcd1234  
• amount: 10  
• version: 1

**ITEM**  
• Id: abcd1234  
• amount: 15  
• version: 1

## Transaction 2

**ITEM**  
• Id: abcd1234  
• amount: 0  
• version: 0

FindById Item

Add amount = 10

Commit Item

Commit Item

Optimistic Locking Exception thrown

FindById Item

Add amount = 5

## DataBase

**ITEM**  
• Id: abcd1234  
• amount: 0  
• version: 0

**ITEM**  
• Id: abcd1234  
• amount: 10  
• version: 1

**ITEM**  
• Id: abcd1234  
• amount: 15  
• version: 2



# Optimistic Locking in JPA

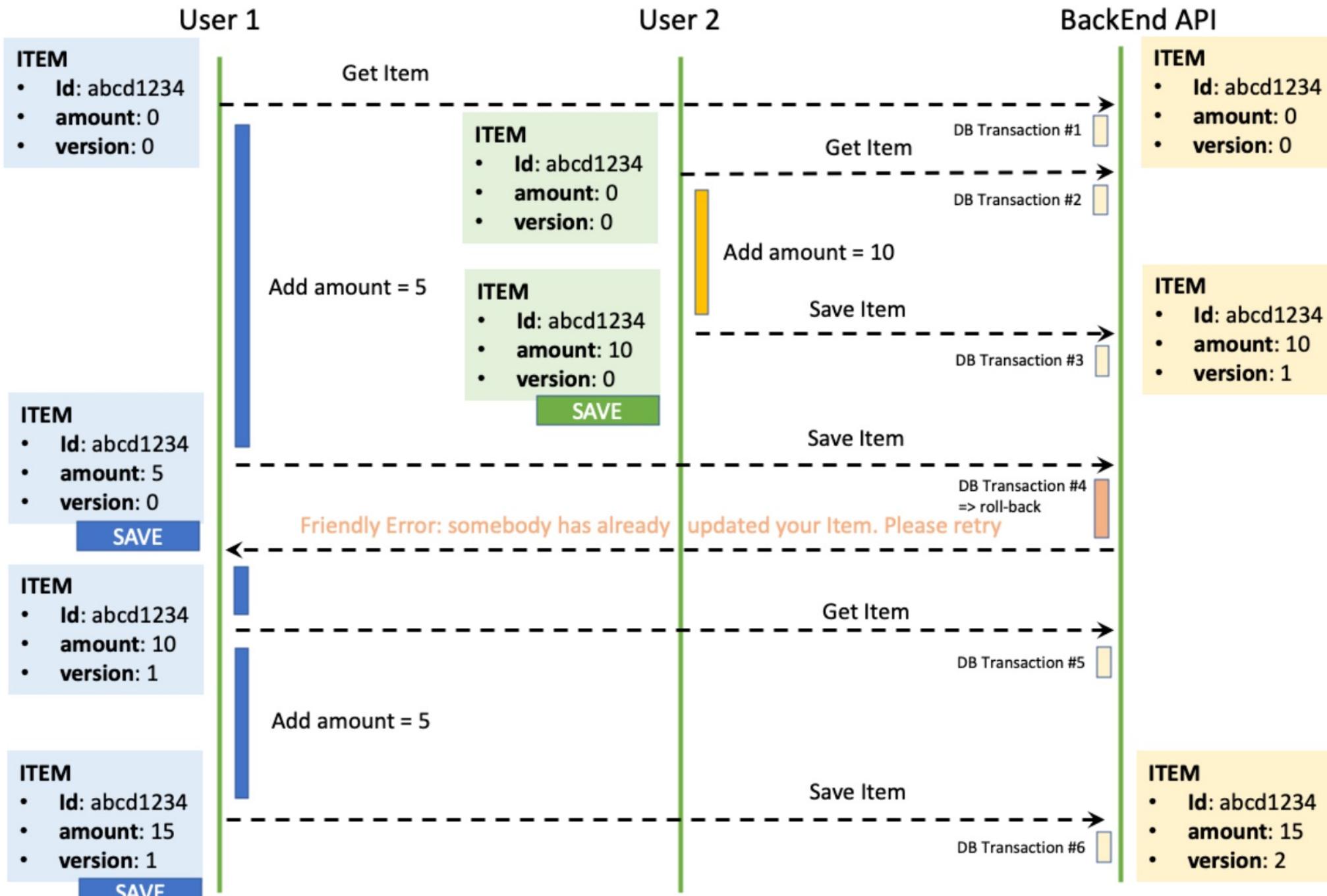
---



```
@Entity  
public class EntityWithVersionField  
{  
    @Version  
    long version;  
}
```

- Wenn ein mit `@Version` annotiertes Attribut vorhanden ist, injiziert JPA automatisch den Versionswert in dieses Feld.
  - Idealerweise verwendet man einen numerischen Typ, etwa ein `long`-Attribut.
  - Besser nicht `TIMESTAMP` wegen Granularität (ms)
  - Versionsfelder sollten von der Anwendung als nur lesend behandelt werden und es sollten keine Mutator-Methoden bereitgestellt werden.
  - **Es ist nur ein Versionsfeld pro Entitätsklasse zulässig.**
-

# Optimistic





- Auch ohne zusätzliche Ausnahmebehandlung hat sich die Situation bereits verbessert: Es würde kein stiller Datenverlust mehr bei Race Conditions auftreten!
- Wie reagiert man am besten auf eine OptimisticLockException?
  - Variante 1 **INFO**: Den Benutzer über eine zwischenzeitliche Änderung informieren. Benutzer muss Daten erneut eingeben (hoffentlich mit Hilfe) und speichern
  - Variante 2 **RETRY**: Exception-Handling mit einfacher / mehrfacher Wiederholung des Aufrufs (idealerweise schon in einem Service gekapselt)
  - Kombination aus beiden: Erst Retry und dann INFO



# Pessimistic Locking





## Pessimistic Lock anfordern

- **PESSIMISTIC\_READ** - die eine Lesesperre darstellt.
- **PESSIMISTIC\_WRITE** - was eine exklusive Sperre darstellt.

```
entityManager.lock(employee, LockModeType.PESSIMISTIC_WRITE);
```

## Pessimistic Lock freigeben

Pessimistic locks werden am Transaktionsende automatisch freigegeben (entweder durch Commit oder Rollback).

---



## transaction #1

## ITEM

- id: abcd1234
- amount: 0

findById item and acquire exclusive lock on it

return item with acquired lock

add amount = 5

commit item and release lock

DB transaction #1 committed

## ITEM

- id: abcd1234
- amount: 5

## transaction #2.x

findById item and acquire exclusive lock on it

waiting time  $\geq$  LockTimeout

Lock timeout exception thrown

DB transaction #2.1 rollbacked

findById item and acquire exclusive lock on it

return item with acquired lock

add amount = 10

commit item and release lock

DB transaction #2.2 committed

## database

## ITEM

- id: abcd1234
- amount: 0



## ITEM

- id: abcd1234
- amount: 5

## ITEM

- id: abcd1234
- amount: 15



# Besonderheiten





## Batch Store

Storing a large number of entity objects requires special consideration. The combination of the `clear` and `flush` methods can be used to save memory in large transactions:

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
    Point point = new Point(i, i);
    em.persist(point);
    if ((i % 10000) == 0) {
        em.flush();
        em.clear();
    }
}
em.getTransaction().commit();
```



Managed entity objects consume more memory than ordinary non managed Java objects. Therefore, holding 1,000,000 managed `Point` instances in the `persistence context` might consume too much memory. The sample code above clears the persistence context after every 10,000 persists. Updates are flushed to the database before clearing, otherwise they would be lost.

Updates that are sent to the database using `flush` are considered temporary and are only visible to the owner `EntityManager` until a commit. With no explicit `commit`, these updates are later discarded. The combination of `clear` and `flush` enables moving the temporary updates from memory to the database.



Updates that are sent to the database using `flush` are considered temporary and are only visible to the owner `EntityManager` until a commit. With no explicit `commit`, these updates are later discarded. The combination of `clear` and `flush` enables moving the temporary updates from memory to the database.

Note: Flushing updates to the database is sometimes also useful [before executing queries](#) in order to get up to date results.

Storing large amount of entity objects can also be performed by multiple transactions:

```
em.getTransaction().begin();
for (int i = 1; i <= 1000000; i++) {
    Point point = new Point(i, i);
    em.persist(point);
    if ((i % 10000) == 0) {
        em.getTransaction().commit();
        em.clear();
        em.getTransaction().begin();
    }
}
em.getTransaction().commit();
```



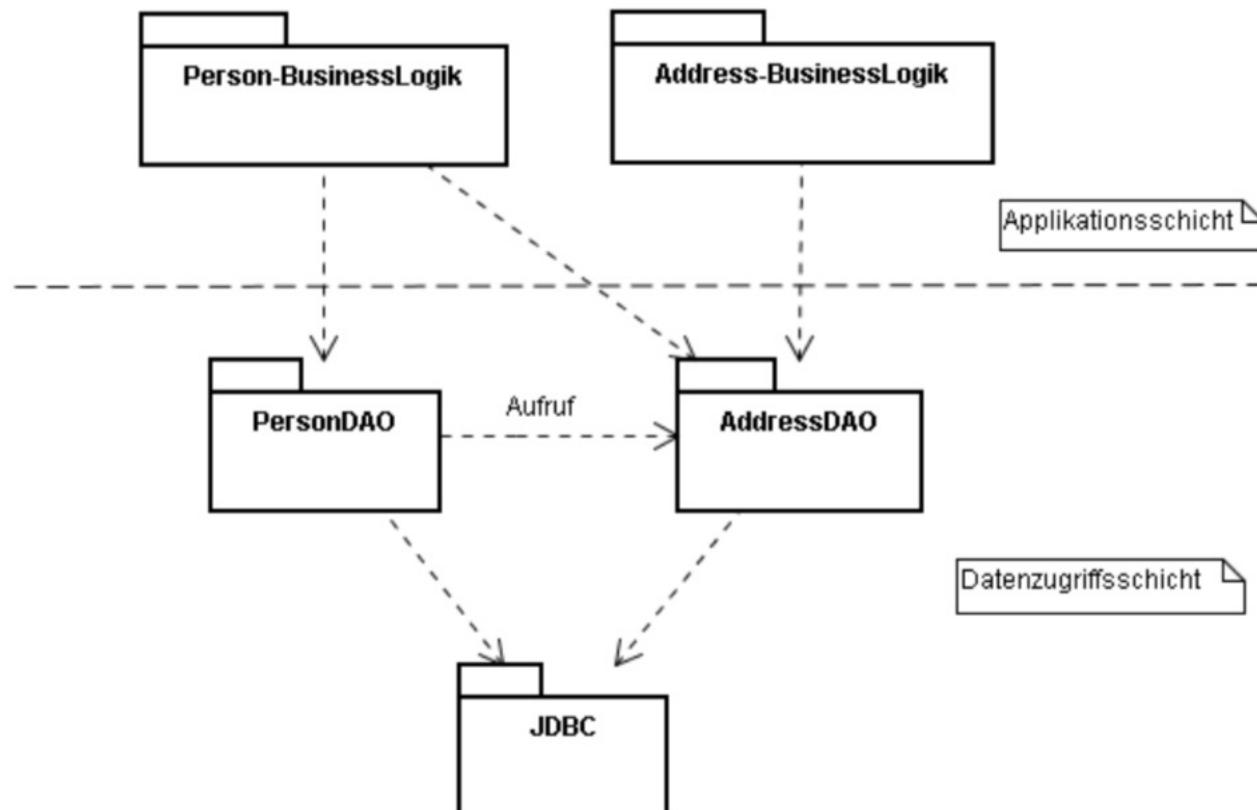
Splitting a batch store into multiple transactions is more efficient than using one transaction with multiple invocations of the `flush` and `clear` methods. So using multiple transactions is preferred when applicable.



# PART 7: DAOs und Repositories



- Das Data Access Object (DAO)-Muster ist ein Strukturmuster
- Es ermöglicht, die Anwendungs-/Geschäftsschicht von der Persistenzschicht mithilfe einer abstrakten API zu isolieren.





- DAO verbirgt alle Komplexitäten, die mit der Durchführung von CRUD-Operationen verbunden sind, vor der Anwendung.
- DAO führt zu loser Kopplung: Bestandteile der Applikation können sich leichter getrennt voneinander weiterentwickeln
- Eng verwandt mit dem Repository Pattern (DAO wird manchmal auch als Repository bezeichnet)
  - DAO ist ein Konzept auf niedrigerer Ebene, näher an der DB.
  - Repository hingegen ist ein eher übergeordnetes Konzept, das a) mehr als CRUD anbietet und auf Collections arbeitet und b) oftmals näher zu Domänenobjekten ist
- Idealerweise erfolgen alle Datenbankzugriffe im System über ein DAO, um eine gute Kapselung zu erreichen.



- Jede DAO-Instanz ist für eine Entität zuständig, insbesondere die CRUD-Operationen: Anlegen, Lesen (nach Primärschlüssel), Aktualisieren und Löschen (CRUD) des Domänenobjekts verantwortlich

```
public interface IPersonDAO
{
    // CRUD-Funktionalität
    List<Person> getAllPersons() throws DataAccessException;
    long addPerson(Person person) throws DataAccessException;
    void updateFromOther(long personId, Person otherPerson) throws DataAccessException;
    void removePerson(long personId) throws DataAccessException;
}
```

- DAO kann weitere Aktionen ermöglichen, etwa spezielle Abfragen
- DAO ist nicht für die Handhabung von Transaktionen, Session oder Connections verantwortlich - diese werden außerhalb der DAO behandelt

# Person DAO



```
public final class PersonDAO
{
    private final EntityManager entityManager;

    public PersonDAO(final EntityManager entityManager)
    {
        this.entityManager = entityManager;
    }

    // C -- CREATE
    public long createPerson(final Person newPerson)
    {
        entityManager.persist(newPerson);
        return newPerson.getId();
    }

    // R -- READ
    public Person findPersonById(final long id)
    {
        return entityManager.find(Person.class, id);
    }
}
```

# Person DAO



```
// R -- READ
public List<Person> findAllPersons()
{
    final TypedQuery<Person> query = entityManager.createQuery("FROM Person",
                                                               Person.class);
    return query.getResultList();
}

// U -- UPDATE
public void updatePersonFrom(final long destId, final Person otherPerson)
{
    final Person personInDb = findPersonById(destId);
    if (personInDb != null)
    {
        personInDb.setFirstName(otherPerson.getFirstName());
        personInDb.setLastName(otherPerson.getLastName());
        personInDb.setBirthday(otherPerson.getBirthday());
    }
}
```

## Person DAO

---



```
// D -- DELETE
public void deletePersonById(final long id)
{
    final Person personInDb = findPersonById(id);
    if (personInDb != null)
    {
        entityManager.remove(personInDb);
    }
}
```

```
private static void executeStatements(final EntityManager entityManager)
{
    // DAO erzeugen
    final PersonDAO dao = new PersonDAO(entityManager);

    // Einfügeoperationen ausführen und das Resultat prüfen
    final Person michael = new Person("Micha-DAO", "Inden", new Date(71, 1, 7));
    final Person michael2 = new Person("Micha-DAO", "Inden", new Date(71, 1, 7));
    final Person werner = new Person("Werner-DAO", "Inden", new Date(40, 0, 31));

    final long michaelId = dao.createPerson(michael);
    final long michaelId2 = dao.createPerson(michael2);
    final long wernerId = dao.createPerson(werner);

    final List<Person> persons2 = dao.findAllPersons();
    persons2.forEach(System.out::println);

    // Änderungen ausführen und das Resultat prüfen
    dao.deletePersonById(michaelId);
    werner.setFirstName("Dr. h.c. Werner");

    final List<Person> persons = dao.findAllPersons();
    persons.forEach(System.out::println);
}
```



- Rudimentäres Interface für einen generischen DAO
  - Save() ist für «create» und «update» zuständig bzw. «update» geschieht durch Attributänderungen

```
import java.util.List;
import java.util.Optional;

public interface Dao<T>
{
    T save(T t);

    T get(long id);
    List<T> getAll();

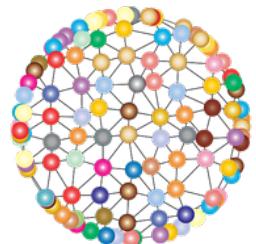
    void delete(T t);
}
```

```
import java.util.List;
import java.util.Optional;

public interface Dao<T>
{
    T save(T t);

    Optional<T> get(long id);
    List<T> getAll();

    void delete(T t);
}
```



**Was haben wir gerade  
aussen vorgelassen?**

# Allgemeineres DAO Pattern

---



- Verschiedene Typen für Keys
- Implementierung und Anbindung an EntityManager

```
public final class GenericDAO<T, K>
{
    private final EntityManager entityManager;
    private final Class<T> clazz;

    GenericDAO(final EntityManager entityManager, final Class<T> clazz)
    {
        this.entityManager = entityManager;
        this.clazz = clazz;
    }

    // C -- CREATE
    public T save(final T newObject)
    {
        entityManager.persist(newObject);
        return newObject;
    }
}
```

# Allgemeineres DAO Pattern

---

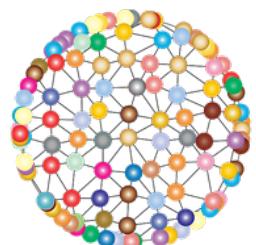


```
// R -- READ
public T findById(final K id)
{
    return entityManager.find(clazz, id);
}

// R -- READ
public List<T> findAll()
{
    final TypedQuery<T> query = entityManager.createQuery("FROM " + clazz.getSimpleName(),
                                                          clazz);
    return query.getResultList();
}

// D -- DELETE
public void deleteById(final K id)
{
    final T objectInDb = findById(id);
    if (objectInDb != null)
    {
        entityManager.remove(objectInDb);
    }
}
```

---



**Das ist schon gut. Wäre es  
nicht super, wenn man das  
noch kürzer und mächtiger  
machen könnte?**



- **Data Access Object (DAO) vereinfachen den Datenzugriff in der Regel enorm**
  - **Einen Schritt weiter gehen die Repositories, die von Spring Data bereitgestellt werden**
  - **Mit Spring Data werden keine DAOs mehr benötigt, stattdessen nur noch Repositories**
  - **Spring Data Repositories bringen von Hause aus bereits eine Sammlung sehr nützlicher Methoden mit**
  - **Eigene Repositories oft mit Basis JpaRepository, zum Teil rein deklarativ**
-



- Mit Spring Data Repositories können eigene Abfragen definiert werden

- Deklarativ

```
public interface IFooDAO extends JpaRepository<Foo, Long> {  
  
    Foo findByName(String name);  
  
}
```

- Mithilfe von JPQL

```
@Query("SELECT f FROM Foo f WHERE LOWER(f.name) = LOWER(:name)")  
Foo retrieveByName(@Param("name") String name);
```



- **Mit Spring Data Repositories können eigene Abfragen definiert werden**

- Mithilfe von JPQL

```
@Query("SELECT u FROM User u WHERE u.status = 1")  
Collection<User> findAllActiveUsers();
```

```
@Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2")  
User findUserByStatusAndName(Integer status, String name);
```

- Mit Native Queries

```
@Query(  
    value = "SELECT * FROM USERS u WHERE u.status = 1",  
    nativeQuery = true)  
Collection<User> findAllActiveUsersNative();
```

# Spring Data Repositories



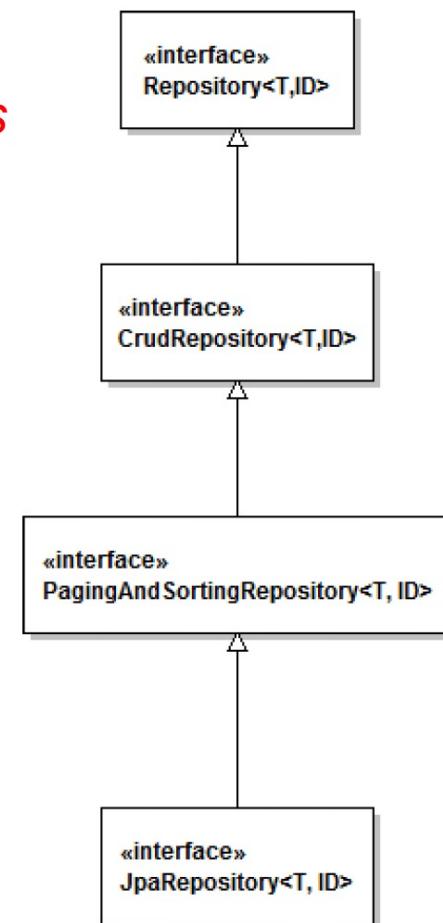
## • Weitere Möglichkeiten

```
@Query(value = "SELECT u FROM User u WHERE u.name IN :names")
List<User> findUserByNameList(@Param("names") Collection<String> names);
```



# Automatic JPA Repositories

- **Repository-Interfaces**
  - `Repository<T, ID>` *marker interface*
  - `CrudRepository<T, ID>` *generic CRUD operations*
    - `<S extends T> S save(S entity)`
    - `Optional<T> findById(ID id);`
    - `Iterable<T> findAll();`
    - `boolean existsById(ID id);`
    - `long count();`
    - `void deleteById(ID id);`
    - `void delete(T entity);`
    - `void deleteAll();`
  - `PagingAndSortingRepository<T, ID>`
    - `Iterable<T> findAll(Sort sort);`
    - `Page<T> findAll(Pageable pageable);`
  - `JpaRepository<T, ID>`





- **Query-Methoden**

```
public interface MovieRepository extends JpaRepository<Movie, Long>
{
    List<Movie> findMovieByTitleIgnoringCase(String title);
}
```

- **Datenbankzugriffe werden basierend auf `findXyz()` automatisch generiert**



## Mögliche Varianten

- **And, Or**
  - `findByLastnameAndFirstname()` / `findByLastnameOrFirstname()`
  - ... where `x.lastname = ?1 and (or) x.firstname = ?2`
- **Is, Equals**
  - `findByFirstnameIs()` / `findByFirstnameEquals()` / `findByFirstname()`
  - ... where `x.firstname = ?1`
- **Between**
  - `findByStartDateBetween()`
  - ... where `x.startDate between ?1 and ?2`
- **LessThan, GreaterThan**
  - `findByAgeLessThan()` / `findByAgeGreaterThan()`
  - ... where `x.age < ?1` / ... where `x.age > ?1`



## Mögliche Varianten

- After, Before
  - IsNull, IsNotNull, NotNull
  - Like / NotLike
  - Containing
  - OrderBy
  - True / False
  - In / NotIn
  - Not
  - IgnoreCase
  - Asc / Desc
- 
- **Begrenzung der Ergebnisgröße einer Abfrage**
    - `findFirstByAddressCityByNameAsc`
    - `findFirst10ByLastnameAsc`



---

## Exercises Part 6 & 7

<https://github.com/Michaeli71/JPA-Workshop.git>





# PART 8: EntityListener & Validation



- **JPA bietet Entity 7 Lifecycle Events**
- **Benachrichtigung über Callback-Methoden, die für Entity-Lifecycle-Ereignisse definiert sind und von JPA automatisch aufgerufen werden, wenn diese Ereignisse eintreten.**
  - **@PrePersist** - before a new entity is persisted (added to the EntityManager).
  - **@PostPersist** - after storing a new entity in the database (during commit or flush).
  - **@PostLoad** - after an entity has been retrieved from the database.
  - **@PreUpdate** - when an entity is identified as modified by the EntityManager.
  - **@PostUpdate** - after updating an entity in the database (during commit or flush).
  - **@PreRemove** - when an entity is marked for removal in the EntityManager.
  - **@PostRemove** - after deleting an entity from the database (during commit or flush).

## Entity Listeners – Variante 1

---



- Direkt an der Entity als Methode(n)

```
@Entity
public class Employee
{
    @PrePersist
    public void logNewEmployeeAttempt() {
        System.err.println("Attempting to add new employee " + this);
    }

    @PostPersist
    public void logNewEmployeeAdded() {
        System.err.println("Added employee '" + this);
    }

    // ...
}
```

# Entity Listeners – Variante 2

---



- **Separate Klasse mit Methoden**

```
public class NoOpListener
{
    @PrePersist void onPrePersist(Object o) {}
    @PostPersist void onPostPersist(Object o) {}
    @PostLoad void onPostLoad(Object o) {}
    @PreUpdate void onPreUpdate(Object o) {}
    @PostUpdate void onPostUpdate(Object o) {}
    @PreRemove void onPreRemove(Object o) {}
    @PostRemove void onPostRemove(Object o) {}
}
```

- **Registrierung an Entity-Klasse**

```
@EntityListeners(NoOpListener.class)
@Entity
public class SomeEntity
{
    // ...
}
```

---

## Entity Listeners – Last Modified aktualisieren



```
@Entity
public class Address
{
    @Id
    private int id;
    private String street;

    private LocalDateTime lastUpdated;

    public void setLastUpdated(LocalDateTime lastUpdated)
    {
        this.lastUpdated = lastUpdated;
    }

    public LocalDateTime getLastUpdated()
    {
        return lastUpdated;
    }

    // ...
}
```



**Wie können wir Last  
Modified automatisch  
aktualisieren?**



## Entity Listeners – Befüllung berechneter Felder



```
@EntityListeners(LastUpdateListener.class)
@Entity
public class Address implements LastUpdateAware
{
    @Id
    private int id;
    private String street;

    private LocalDateTime lastUpdated;

    // Befüllung durch Listener
    @Override
    public void setLastUpdated(LocalDateTime lastUpdated)
    {
        this.lastUpdated = lastUpdated;
    }

    // ...
}
```

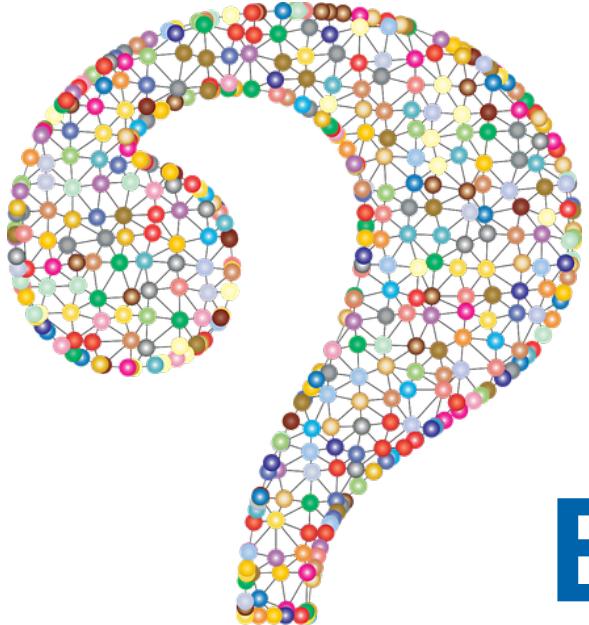
## Entity Listeners – Befüllung berechneter Felder

---

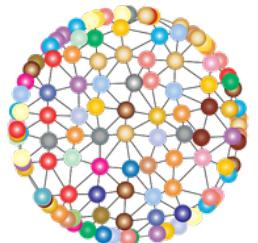


```
public interface LastUpdateAware
{
    public void setLastUpdated(LocalDateTime lastUpdated);
}

public class LastUpdateListener
{
    /**
     * automatic property set before any database persistence
     */
    @PreUpdate
    @PrePersist
    public void setLastUpdate(LastUpdateAware destinationObj)
    {
        destinationObj.setLastUpdated(LocalDateTime.now());
    }
}
```



**Wie können wir  
Berechnungen ausführen?**



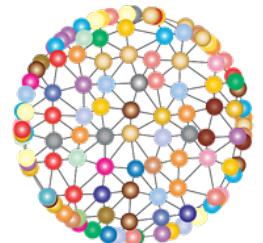
# Entity Listeners – Befüllung „berechneter“ Felder beim Laden



```
@Entity
public class Address implements LastUpdateAware
{
    @Id
    private int id;
    private String street;
    // ...
    @Transient
    private long syncTimeInMs;

    @PostPersist
    @PostUpdate
    @PostLoad
    public void resetSyncTime()
    {
        syncTimeInMs = System.currentTimeMillis();
    }

    public long getTimeSinceLastSyncInMs()
    {
        return System.currentTimeMillis() - syncTimeInMs;
    }
    // ...
}
```



**Was gibt es sonst  
noch Wissenswertes?**



- Es können auch mehrere Listener registriert werden

```
@Entity  
@EntityListeners({ FirstListener.class, SecondListener.class })  
public class MyEntityWithTwoListeners  
{  
    // ...  
}
```

- Listener werden an ihre Unterklassen vererbt, Subklasse kann dies explizit mit der Annotation @ExcludeSuperclassListeners verhindern:

```
@Entity  
@ExcludeSuperclassListeners  
public class EntityWithNoListener extends MyEntityWithTwoListeners  
{  
    // ...  
}
```



- **Vererbung**

Standardmäßig wird eine Callback-Methode in einer Superentitätsklasse auch für Entitätsobjekte der Unterklassen aufgerufen, es sei denn, diese Callback-Methode wird von der Unterklasse überschrieben.

- **Keine EntityManager- / Query-Methoden**

Um Konflikte mit der DB-Operation zu vermeiden, die das Entity-Lifecycle-Ereignis auslöst (das noch in Bearbeitung ist), sollten Callback-Methoden weder EntityManager- oder Query-Methoden aufrufen noch auf andere Entities zugreifen.

- **Exceptions**

Wenn eine Callback-Methode eine Exception innerhalb einer aktiven Transaktion auslöst, wird die Transaktion für ein Rollback markiert und es werden keine weiteren Callback-Methoden für diese Operation aufgerufen.

---



---

# DEMO

**EntityListenerExample.java**

---



---

# Validation



# Validation

---



```
import javax.validation.constraints.AssertTrue;
import javax.validation.constraints.Email;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class User
{
    @NotNull(message = "Name cannot be null")
    private String name;

    @AssertTrue
    private boolean working;

    // ...
}
```

## Validation „pur“



```
public class User
{
    @NotNull(message = "Name cannot be null")
    private String name;

    @AssertTrue
    private boolean working;

    @Size(min = 10, max = 200,
          message = "About Me must be between 10 and 200 characters")
    private String aboutMe;

    @Min(value = 18, message = "Age should not be less than 18")
    @Max(value = 150, message = "Age should not be greater than 150")
    private int age;

    @Email(message = "Email should be valid")
    private String email;

    // standard setters and getters
}
```

# Validation „pur“



```
public class ProgrammaticValidationExample
{
    public static void main(final String[] args)
    {
        UserWithValidation user = new UserWithValidation();
        user.setWorking(false);
        user.setAboutMe("No info about me!");
        user.setAge(11);

        try (ValidatorFactory factory = Validation.buildDefaultValidatorFactory())
        {
            Validator validator = factory.getValidator();
            Set<ConstraintViolation<UserWithValidation>> violations = validator.validate(user);
            for (ConstraintViolation<UserWithValidation> violation : violations)
            {
                System.err.println(violation.getMessage());
            }
        }
    }
}
```

Name cannot be null  
muss wahr sein  
Age should not be less than 18

# Validation in JPA



```
@Entity
public class UserWithValidation
{
    @Id @GeneratedValue
    private Long id;

    @NotNull(message = "Name cannot be null")
    private String name;

    @AssertTrue
    private boolean working;

    @Size(min = 10, max = 200,
          message = "About Me must be between 10 and 200 characters")
    private String aboutMe;

    @Min(value = 18, message = "Age should not be less than 18")
    @Max(value = 150, message = "Age should not be greater than 150")
    private int age;

    // ...
    // standard setters and getters
}
```

## Validation in JPA



```
private static void executeStatements(final EntityManager entityManager)
{
    UserWithValidation user = new UserWithValidation();
    user.setWorking(false);
    user.setAboutMe("No info about me!");
    user.setAge(11);

    entityManager.persist(user);
    System.out.println(user);
}

ConstraintViolationImpl{interpolatedMessage='Age should not be less than 18',
propertyPath=age, rootBeanClass=class t_validation.UserWithValidation,
messageTemplate='Age should not be less than 18'}
ConstraintViolationImpl{interpolatedMessage='muss wahr sein', propertyPath=working,
rootBeanClass=class t_validation.UserWithValidation,
messageTemplate='{javax.validation.constraints.AssertTrue.message}'}
ConstraintViolationImpl{interpolatedMessage='Name cannot be null', propertyPath=name,
rootBeanClass=class t_validation.UserWithValidation, messageTemplate='Name cannot be
null'}
```



# DEMO

**ValidationExample.java**

---

## Validation -- Annotations

---



- **@NotNull**      **validiert, dass das Attribut nicht null ist.**
- **@AssertTrue**    **prüft, dass der Attributwert true ist.**
- **@Size**            **prüft, dass der Attributwert zwischen den Attributen min und max liegt; für Strings, Collections, Maps und Arrays verfügbar**
- **@Min**            **prüft, dass der Attributwert einen Wert hat, der nicht kleiner als das Attribut value ist.**
- **@Max**            **prüft, dass die kommentierte Eigenschaft einen Wert hat, der nicht größer als das Attribut value ist.**
- **@Email**           **validiert, dass dies eine gültige E-Mail-Adresse ist.**
- **@NotEmpty**     **überprüft, dass das Attribut nicht null oder leer ist; für String-, Collection-, Map- oder Array-Werte anwendbar.**

## Validation -- Annotations



- **@NotBlank** kann nur auf Textwerte angewendet werden und validiert, dass die Eigenschaft nicht Null oder Leerzeichen ist.
- **@Positive & @PositiveOrZero** werden auf numerische Werte angewendet und prüfen, ob diese positiv oder positiv einschließlich 0 sind.
- **@Negative & @NegativeOrZero** gelten für numerische Werte und bestätigen, dass sie negativ oder negativ, einschließlich 0, sind.
- **@Past & @PastOrPresent** prüfen, ob ein Datumswert in der Vergangenheit oder in der Vergangenheit einschließlich der Gegenwart liegt; für alle Datumstypen einschließlich der in Java 8
- **@Future & @FutureOrPresent** fordern, dass ein Datumswert in der Zukunft oder in der Zukunft einschließlich der Gegenwart liegt.

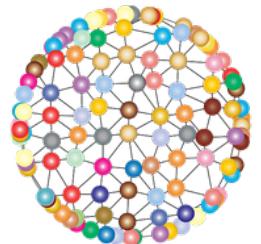


## \* Anpassungen in Persistence Unit

<validation-mode>AUTO</validation-mode>

<validation-mode>CALLBACK</validation-mode>

- Fallstricke Versionen Hibernate & Hibernate-Validator sowie javax.validation / jakarta.validation
  - **Hibernate-Validator 7.x ⇔ / jakarta.validation**
  - **Hibernate-Validator 6.x ⇔ / javax.validation**
- Nur ältere Variante läuft in Persistence Unit sauber, ansonsten Versions- und Initialisierungsprobleme
- Standalone geht beides problemlos



# Wie kann man eigene Validatoren bauen?

# Eigene Validatoren in JPA

---



```
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = CheckEnumValidator.class)
public @interface CheckEnum
{
    String message() default "Please enter a valid enum value for this field.";
    Class<?> type();

    // für Constraint
    Class<?>[] groups() default {};
    Class<?extends Payload>[] payload() default {};
}
```

# Eigene Validatoren in JPA

---



```
public class CheckEnumValidator implements ConstraintValidator<CheckEnum, String>
{
    Class<?> type;

    @Override
    public void initialize(CheckEnum constraintAnnotation)
    {
        type = constraintAnnotation.type();
        if (!type.isEnum())
            throw new IllegalArgumentException("type is not an enum");
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context)
    {
        ...
    }
}
```

# Eigene Validatoren in JPA



```
public class CheckEnumValidator implements ConstraintValidator<CheckEnum, String>
{
    ...

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context)
    {
        if (value == null)
            return true;

        Enum<?>[] enumValues = (Enum<?>[])type.getEnumConstants();
        for (Enum<?> enumValue : enumValues)
        {
            if (enumValue.name().equals(value.trim()))
                return true;
        }
        return false;
    }
}
```

# Eigene Validatoren in JPA



```
public class ValidatedDomainClass
{
    @NotNull(message = "Deposit Date is required.")
    @CheckLocalDate(dateFormat = { "yyyy-MM-dd" })
    String depositDate;

    @CheckLocalDate(dateFormat = "dd.MM.yyyy")
    String publicationDate;
    @CheckLocalDate(dateFormat = { "dd.MM.yyyy", "dd.MM.yy" })
    String collectionDate;

    // Enum-Validator für Legacy
    @CheckEnum(type = Seasons.class)
    String season;
    @CheckEnum(type = SpecialColors.class)
    String color;
    @CheckListOfValues(allowedValues = { "Anne", "Will", "Peter", "Lustig" })
    String value;
}
```



# DEMO

**CustomValidatorsExample.java**

---



# Part 9: Caching





- **JPA unterstützt im Standard sowohl 1st (L1) als auch 2nd (L2) Level Caches:**
  - L1 ist standardmäßig aktiv und vom JPA-Provider bereitgestellt
  - L2 muss in der Persistence Unit durch Parametrierung aktiviert werden
  - L2 muss extern eingebunden werden
- **Caching ist gerade im Kontext von Datenbanken hilfreich, um „teure“ Datenbankabfragen zu reduzieren**
- **Signifikante Performancesteigerungen möglich**
- **Nahezu transparent, Applikation nutzt EntityManager / JPA wie zuvor**
- **Birgt aber auch Fallstricke, wie Stale Data und verbraucht Hauptspeicher**



## 1st Level Cache -- EntityManager

- Jeder **EntityManager** besitzt einen Persistenzkontext.
- Der **Persistenzkontext** dient als 1st Level Cache (L1 Cache)
- Der **Persistenzkontext** ist eine Sammlung aller Entitätsobjekte ist, die der **EntityManager** verwaltet..
- Ein Versuch, ein Entitätsobjekt aus der DB zu lesen, das bereits vom EntityManager verwaltet wird, gibt die vorhandene Instanz aus dem Persistenzkontext zurück.
- Zugriff: L1 (-> L2) -> DB

## 2nd Level Cache -- EntityManagerFactory

- verwaltet viel mehr Entitätsobjekten, wird von allen EntityManager-Objekten gemeinsam genutzt/gefüllt
- Wenn Entitäten nicht im Persistenzkontext gefunden werden, werden sie aus dem L2-Cache geladen, falls sie dort vorhanden sind
- ~~Zugriff: L1 -> L2 -> DB~~



## Zusammenspiel

- Typische Webapplikation mit vielen kurzen Transaktionen (z.B. per Request).
- Dadurch keine gemeinsame Nutzung von häufig verwendeten Daten möglich.
- Als Folge hilft ein L1 Cache nicht zur performance-Optimierung.
- Ein L2 Cache erlaubt es, Daten auf einer anderen Transaktion direkt ohne DB-Zugriff aus dem Cache zu verarbeiten.
- Zugriffe: L1 -> L2 -> DB

## Entities für Caching vormerken / davon ausschließen



Die Annotation **@Cacheable** erlaubt eine feingranulare Steuerung des Cachings auf Basis von Typen

```
@Cacheable // or @Cacheable(true)  
@Entity  
public class MyCacheableEntityClass  
{  
    // ...  
}
```

```
@Cacheable(false)  
@Entity  
public class MyNonCacheableEntityClass extends MyCacheableEntityClass  
{  
    /// ...  
}
```



- **EHCache ist ein weitverbreiteter L2 Cache, perfekt in Hibernate integriert**
- **Maven-Dependency**

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.4.30.Final</version>
</dependency>
```

- **Gradle Dependency**

```
// https://mvnrepository.com/artifact/org.hibernate/hibernate-ehcache
implementation group: 'org.hibernate', name: 'hibernate-ehcache', version: '5.4.30.Final'
```

- **Weitere Infos:**

<https://www.ehcache.org/>

---

## Cache Modi konfigurieren

---



- **ALL** – Alle Entities werden im Second Level Cache gehalten.
  - **NONE** – Es werden keine Entities für die Persistence Unit gecacht.
  - **ENABLE\_SELECTIVE** – Es werden nur solche Entities gecacht, die mit der Annotation `@Cacheable` bzw. `@Cacheable(true)` markiert sind.
  - **DISABLE\_SELECTIVE** – Es werden alle Entities gecacht, bis auf diejenigen, die explizit mit der Annotation `@Cacheable(false)` davon ausgeschlossen wurden.
  - **UNSPECIFIED** – Das Verhalten ist nicht spezifiziert und es wird der Default vom verwendeten JPA-Provider genutzt.
- 
- **ENABLE\_SELECTIVE** fast immer eine recht vernünftige Wahl dar, weil hierbei all diejenigen Entities gecacht werden, für die man dies explizit vorgegeben hat.
  - Die zweitbeste Wahl ist wohl **DISABLE\_SELECTIVE**, weil hier einige Entitäten explizit vom Caching ausgeschlossen werden.

# Cache-Konfiguration in Persistence Unit



```
<persistence-unit name="java-profi-PU-CACHING" transaction-type="RESOURCE_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
<class>y_caching.Person</class>
...
<exclude-unlisted-classes>true</exclude-unlisted-classes>

<!-- <shared-cache-mode>ALL</shared-cache-mode> -->
<!-- <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode> -->
<!-- <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode> -->
<shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>

<properties>
  ...
  <property name="hibernate.cache.use_second_level_cache" value="true"/>
  <property name="hibernate.cache.region.factory_class"
            value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
  ...

```



---

# Das Interface Cache



# Das Interface Cache



```
public interface Cache  
{  
    public boolean contains(Class cls, Object primaryKey);  
    public void evict(Class cls, Object primaryKey);  
    public void evict(Class cls);  
    public void evictAll();  
}
```

- **Zugriff erhält man von der EntityManagerFactory oder dem EntityManager:**

```
Cache cache = entityManagerFactory.getCache();
```

```
Cache cache = entityManager.getEntityManagerFactory().getCache();
```

- **ABER: Die Interface Cache und die Methoden sind in vielen Anwendungen eher unnötig.**

## Beispiel: Existenz im Cache prüfen

---



```
Cache cache = entityManager.getEntityManagerFactory().getCache();

if (cache.contains(Person.class, 1L))
{
    System.out.println("Person with id 1 found in cache");
}
else
{
    System.out.println("Person with id 1 NOT in cache");
}
```

## Beispiel: Daten aus dem Cache entfernen

---



```
// manually evict user form the second-level cache  
cache.evict(Person.class, 1L);  
  
// Remove all the instances of a specific class from the cache:  
cache.evict(Person.class);  
  
// Clear the shared cache by removing all the cached entity objects:  
cache.evictAll();
```

## Das Cache Interface – GröÙe typabhängig abfragen

---



```
import net.sf.ehcache.CacheManager;
```

```
Cache cache = entityManager.getEntityManagerFactory().getCache();
System.out.println("Cache size: " + getCacheSize(Person.class));
```

```
private static int getCacheSize(Class<?> clazz)
{
    return CacheManager.ALL_CACHE_MANAGERS.get(0).getCache(clazz.getName()).getSize();
}
```

---

## Beispielaktionen



```
Cache cache = entityManager.getEntityManagerFactory().getCache();
System.out.println("Cache size: " + getCacheSize(Person.class));

if (cache.contains(Person.class, 1L))
{
    if (cache.contains(Person.class, 2L))
    {
        // manually evict user from the second-level cache
        cache.evict(Person.class, 2L);
    }
    else
    {
        System.out.println("No Person with id 2 in cache");
    }
}
else
{
    System.out.println("No Person with id 1 in cache");
}
```

## Beispielaktionen



```
System.out.println("Size after 1: " + getCacheSize(Person.class));  
  
// manually evict user from the second-level cache  
cache.evict(Person.class, 1L);  
System.out.println("Size after 2: " + getCacheSize(Person.class));  
  
// Remove all the instances of a specific class from the cache:  
cache.evict(Person.class);  
System.out.println("Size after 3: " + getCacheSize(Person.class));  
  
// Clear the shared cache by removing all the cached entity objects:  
cache.evictAll();  
System.out.println("Size after 4: " + getCacheSize(Person.class));
```

```
Cache size: 4  
Size after 1: 3  
Size after 2: 2  
Size after 3: 0  
Size after 4: 0
```



# DEMO

**PersonCachingExample.java**

---



## Vorteile

- vermeidet Datenbankzugriffe für bereits geladene Entitäten
- schnelleres Lesen häufig genutzter, unveränderter Entitäten

## Nachteile des L2-Cachings

- Speicherverbrauch für große Mengen an Objekten
- Veraltete Daten für aktualisierte Objekte
- Schlechte Skalierbarkeit für häufig oder gleichzeitig aktualisierte Entitäten

## Daher L2-Caching für Entitäten konfigurieren, die:

- häufig gelesen werden
- selten geändert werden
- Nicht (so) kritisch, wenn veraltet



---

## Exercises Part 8 & 9

<https://github.com/Michaeli71/JPA-Workshop.git>

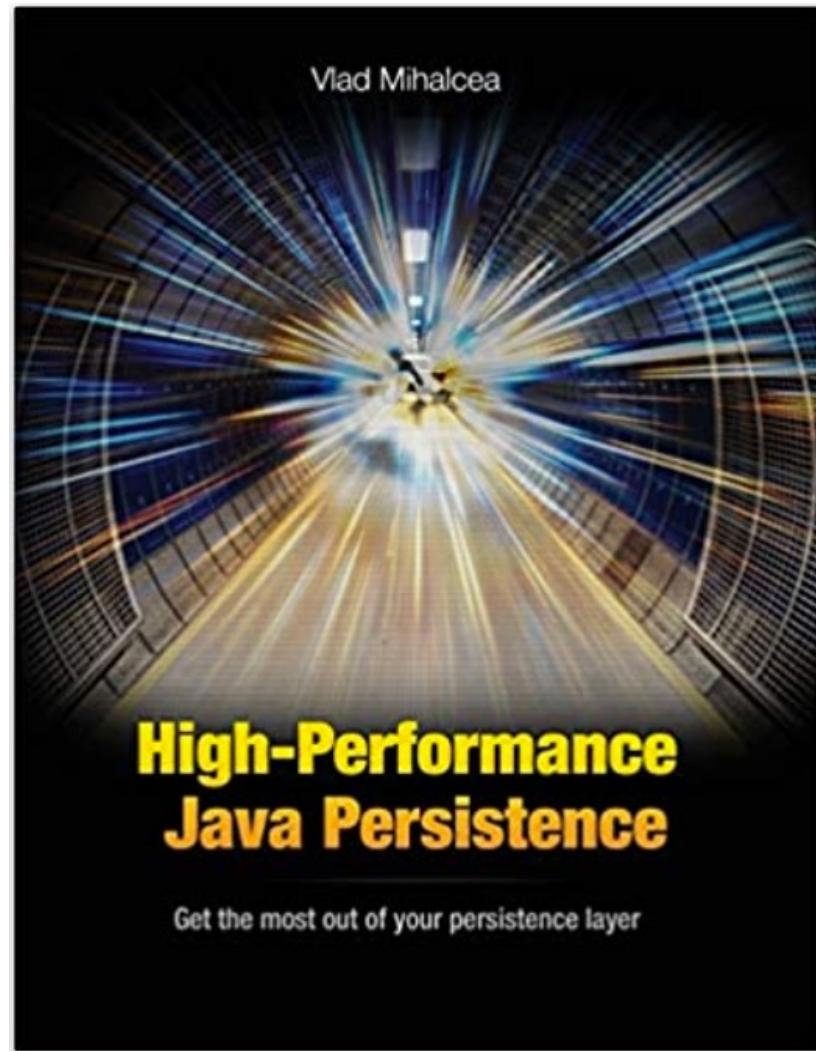
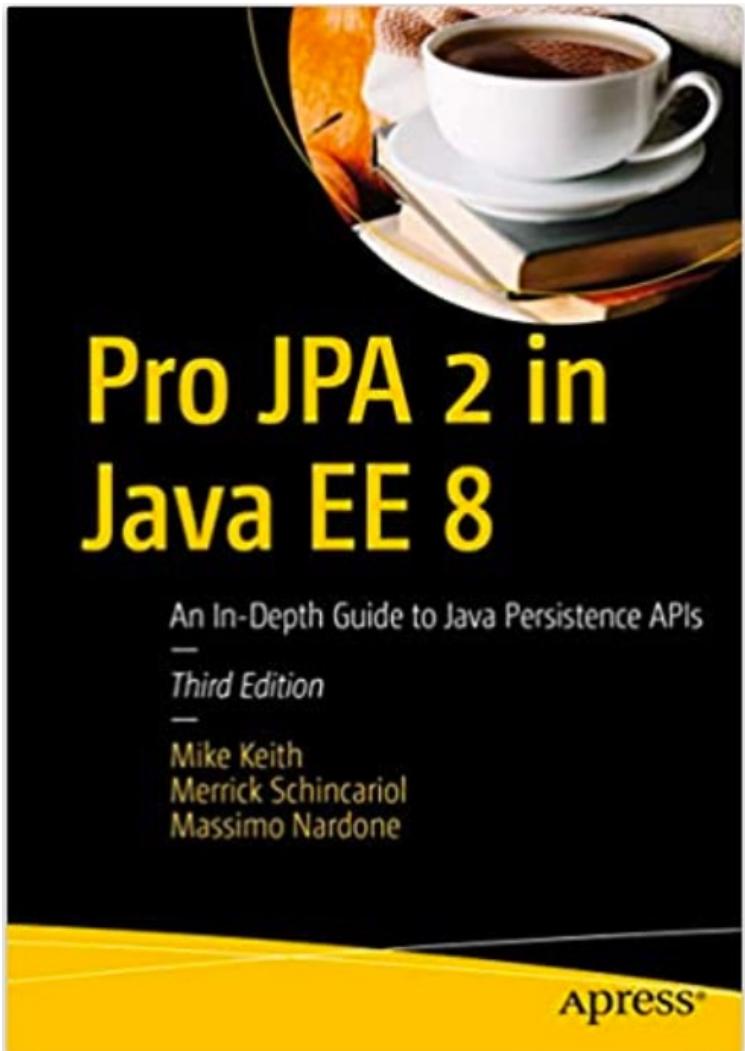




# Questions?



## Empfehlenswerte Bücher



## Weitere Infos / Quellen

---



- **ORM**

- <https://thorben-janssen.com/jpa-generate-primary-keys/>
  - <https://www.objectdb.com/java/jpa/entity/generated>
  - <https://vladmirhalcea.com/orphanremoval-jpa-hibernate/>
  - <https://www.baeldung.com/jpa-one-to-one>
  - <https://www.baeldung.com/jpa-cascade-remove-vs-orphanremoval>
  - <https://www.baeldung.com/hibernate-inheritance>
  - <https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/>
  - <https://www.objectdb.com/api/java/jpa/MappedSuperclass>
  - <https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/mapped-super-class.html>
  - <https://vladmirhalcea.com/the-best-way-to-map-a-onetoone-relationship-with-jpa-and-hibernate/>
  - <https://www.baeldung.com/jpa-many-to-many>
  - <https://vladmirhalcea.com/the-best-way-to-use-the-manytomany-annotation-with-jpa-and-hibernate/>
  - <https://stackabuse.com/a-guide-to-jpa-with-hibernate-relationship-mapping/>
  - <https://thorben-janssen.com/best-practices-for-many-to-many-associations-with-hibernate-and-jpa/>
-



- **JPQL & Criteria API & JOIN FETCH**
  - <https://www.objectdb.com/java/jpa/query>
  - <https://thorben-janssen.com/criteria-updatedelete-easy-way-to/>
  - <https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/fetch-join.html>
  - <https://thorben-janssen.com/hibernate-tips-difference-join-left-join-fetch-join/>
  - <https://dzone.com/articles/how-to-decide-between-join-and-join-fetch>
- **Converter & Date and Time API**
  - <https://thorben-janssen.com/jpa-21-how-to-implement-type-converter/>
  - <https://thorben-janssen.com/persist-localdate-localdatetime-jpa/>
  - <https://thorben-janssen.com/map-date-time-api-jpa-2-2/>
  - <https://www.baeldung.com/jpa-java-time>



- **EntityGraph**
    - <https://www.baeldung.com/jpa-entity-graph>
    - <https://www.baeldung.com/spring-data-jpa-named-entity-graphs>
    - <https://thorben-janssen.com/jpa-21-entity-graph-part-1-named-entity/>
    - <https://turkogluc.com/understanding-jpa-entity-graphs/>
    - <https://turkogluc.com/understanding-the-effective-data-fetching-with-jpa-entity-graphs-part-2/>
    - <https://blog.doubleslash.de/entitygraphs-dynamischer-performanceschub-bei-datenbankabfragen-mit-jpa/>
    - <https://docs.oracle.com/javaee/7/tutorial/persistence-entitygraphs001.htm>
  - **Transactions**
    - <https://www.youtube.com/watch?v=SUQxXg229Xg>
-

## Weitere Infos / Quellen

---



- **Entity Listeners**
  - <https://www.baeldung.com/jpa-entity-lifecycle-events>
  - <https://www.objectdb.com/java/jpa/persistence/event>
- **Validation**
  - <https://www.baeldung.com/javax-validation>
  - [https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html\\_single/](https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/)



- **Caching**
  - <https://www.objectdb.com/java/jpa/persistence/cache>
  - <https://www.youtube.com/watch?v=G6Y5wDF6h5Q>
  - <https://www.ehcache.org/>
  - <https://www.ehcache.org/documentation/2.8/configuration/configuration.html>
  - <http://www.thejavageek.com/2014/09/25/jpa-caching-example/>
- **Performance**
  - [https://www.youtube.com/watch?v=F5asq8ZG\\_1k](https://www.youtube.com/watch?v=F5asq8ZG_1k)
  - [https://www.youtube.com/watch?v=fKZnw\\_3D-5c](https://www.youtube.com/watch?v=fKZnw_3D-5c)
  - <https://www.youtube.com/watch?v=FUt1VM1gNkg>
  - [https://www.youtube.com/watch?v=48w\\_8i9mCoM](https://www.youtube.com/watch?v=48w_8i9mCoM)



- **Allgemeinere Tutorials**

- <https://www.javatpoint.com/jpa-tutorial>
- <https://www.objectdb.com/java/jpa>
- [https://www.youtube.com/watch?v=GKIXI\\_Vc28k](https://www.youtube.com/watch?v=GKIXI_Vc28k)
- <https://howtodoinjava.com/hibernate-tutorials/>

- **Schema Generation**

- <https://thorben-janssen.com/standardized-schema-generation-data-loading-jpa-2-1/>



# Thank You