

# Java Persistence API

# Grundlagen

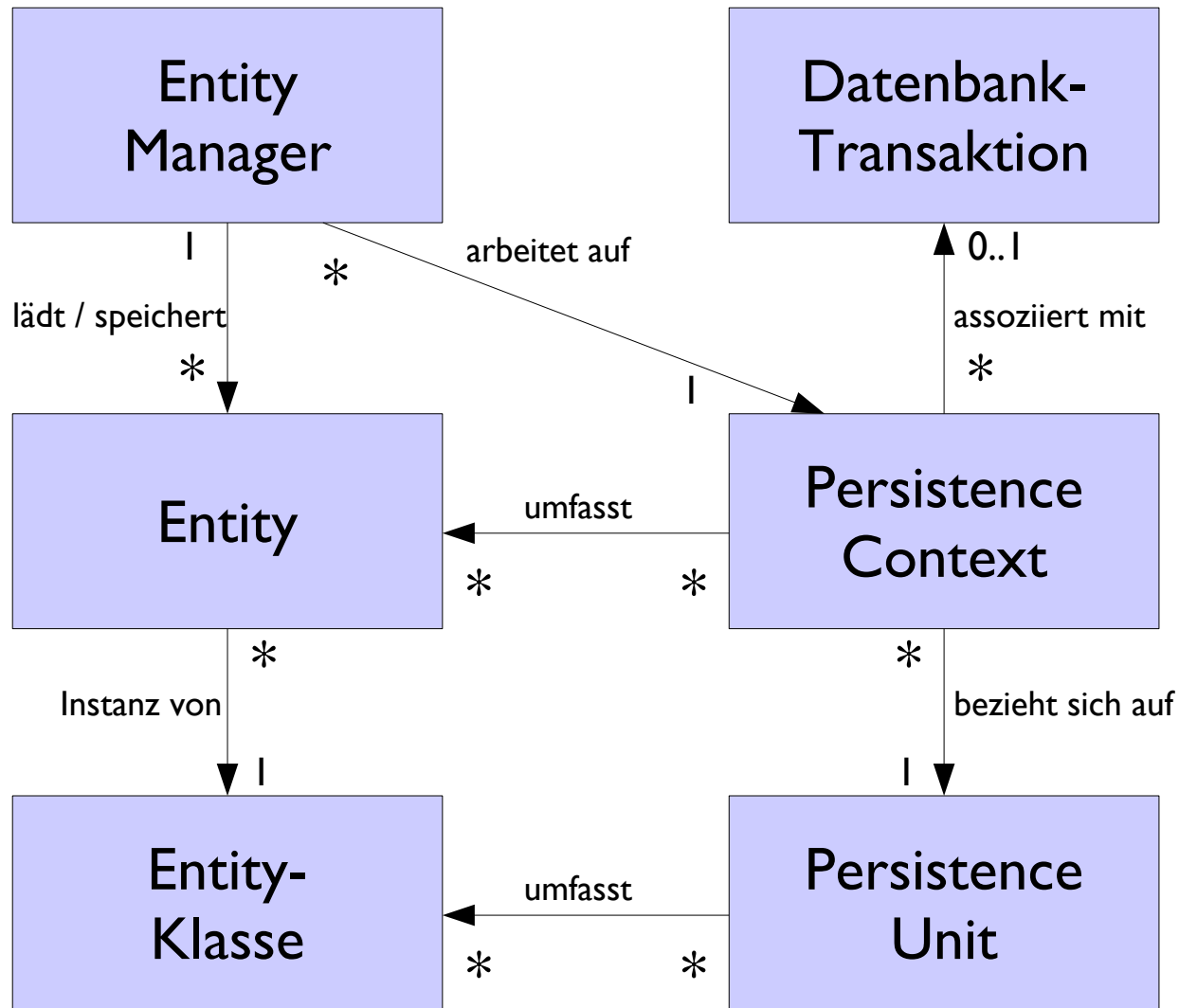
- ▼ Eigene Spezifikation seit Java EE 5
  - Java Persistence API 1.0
  - Abstraktion über JDBC
- ▼ Spezifikation in JEE 6/7/8
  - Java Persistence API 2.0/2.1/2.2
- ▼ Standard für Object-to-Relational-Mapping (ORM)
  - JPA kann Java Objekte automatisch auf eine relationale Datenbank abbilden
- ▼ Bereitstellung einer Abfragensprache (JPQL), die mit Java Objekten arbeitet
- ▼ In JPA spielt **`javax.persistence.EntityManager`** eine der wichtigsten Rollen

- ▼ Jakarta EE 8
  - ▼ JPA 2.2
- ▼ Jakarta EE 9
  - ▼ JPA 3.0
  - ▼ package ***jakarta.persistence***
- ▼ Jakarta EE 10
  - ▼ JPA 3.1
- ▼ Jakarta EE 11
  - ▼ JPA 3.2

- ▶ Leichtgewichtiges Domänenobjekt
- ▶ Persistentes Objekt
- ▶ JavaBeans Klasse
  - mit einigen Regeln
- ▶ Plain Old Java Object (POJOs)
- ▶ Der Operator `new( )`
  - alloziert Entities
  - `new Customer();`

```
@Entity
public class Customer {
    private int id;
    private String name;

    @Id @GeneratedValue
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String n) {
        this.name = n;
    }
}
```



## ▼ Persistence Context

- Menge von attached/managed Objekten
- wird von EntityManager verwaltet
- wenn ein Persistence Context geschlossen ist, werden alle Managed Persistence Objekte als
  - Detached und
  - Unmanaged markiert

## ▼ Typen von Persistence Contexts

- **Transaction-Scoped**
  - Existiert für die Dauer einer Transaktion
- **Extended**
  - Existiert über Transaktionsgrenzen hinweg

- ▶ Applikation Server Managed Persistence Contexts können Transaction-Scoped sein.
- ▶ EntityManager Instanzen injizieren mit dem
  - `@javax.persistence.PersistenceContext` oder
  - seiner ***XML*** äquivalenten Beschreibung
  - können Transaction-Scoped sein

```
@javax.persistence.PersistenceContext(unitName="carDatabase")
private javax.persistence.EntityManager em;

@Transactional(REQUIRED)
public User updateCustomer(long id, String firstName){
    // JTA Transaktion fängt an
    User u = em.find(User.class, id);
    u.setFirstName("new name");
    return u;
    // JTA Transaktion endet
}
```



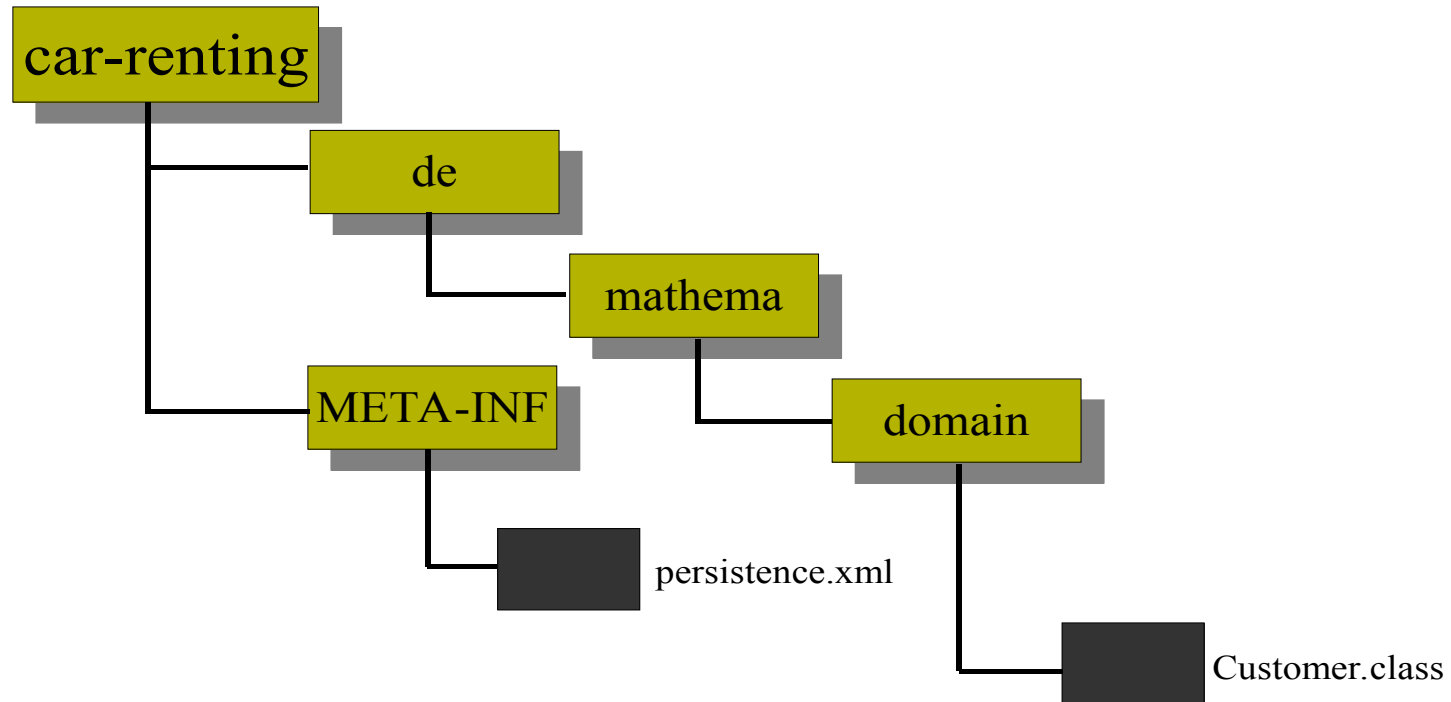
```
@PersistenceContext(unitName="carDatabase", type=EXTENDED)
private javax.persistence.EntityManager em;
...
User user = null;
getTransaction().begin(); //JTA beginnt Transaktion 1
user = em.find(User.class, 1);
getTransaction().commit(); //JTA beendet Transaktion 1

getTransaction().begin(); //JTA beginnt Transaktion 2
user.setFirstName("Serge");
em.flush();
getTransaction().commit(); // User Instanz bleibt managed
                           // und Änderung wird synchronisiert
```

- ▶ Persistence Contexts können auch länger als eine Transaktion leben
- ▶ EntityManager behält den gleichen Persistence Context für seinen ganzen Lebenszyklus
- ▶ Extended Persistence Context kann nur in eine **Stateful Session Bean** injiziert werden


## ▼ Persistence-Unit

- Menge von Klassen, die auf eine bestimmte Datenbank abgebildet werden
- wird in einer Datei persistence.xml definiert
  - Anforderung für die Java Persistence Spezifikation
  - **persistence.xml** definiert eine oder mehrere Persistence-Units
- Diese Datei befindet sich im **META-INF/** Verzeichnis von:
  - einer **einfachen JAR Datei** im CLASSPATH eines regulären Java SE Programms
  - einer **EJB-JAR Datei**: Persistence-Unit kann mit einem EJB Deployment eingefügt werden
  - einer **JAR Datei** in dem **WEB-INF/lib** Verzeichnis in einer Web Archivdatei (.war)
  - einer **JAR Datei** in der Wurzel einer Enterprise Archivdatei (.ear)
  - einer **JAR Datei** in dem **EAR lib/** Verzeichnis



```
public class Persistence {  
    public static EntityManagerFactory createEntityManagerFactory(  
        String unitName);  
    public static EntityManagerFactory createEntityManagerFactory(  
        String persistenceUnitName, java.util.Map properties);  
}
```

```
EntityManagerFactory factory =  
    createEntityManagerFactory("carDatabase");  
...  
EntityManager em = factory.createEntityManager();  
...  
factory.close();
```



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
  <persistence-unit name="carDatabase">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto"
value="create-drop" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

- ▼ Die `createEntityManager()` gibt eine `EntityManager` Instanz zurück
- ▼ Der **Map** Parameter überschreibt oder erweitert die Eigenschaften einer ***persistence.xml*** Datei
- ▼ `EntityManagerFactory.close()` schließt die Factory
- ▼ `EntityManagerFactory.isOpen()` überprüft die Validität der Factory

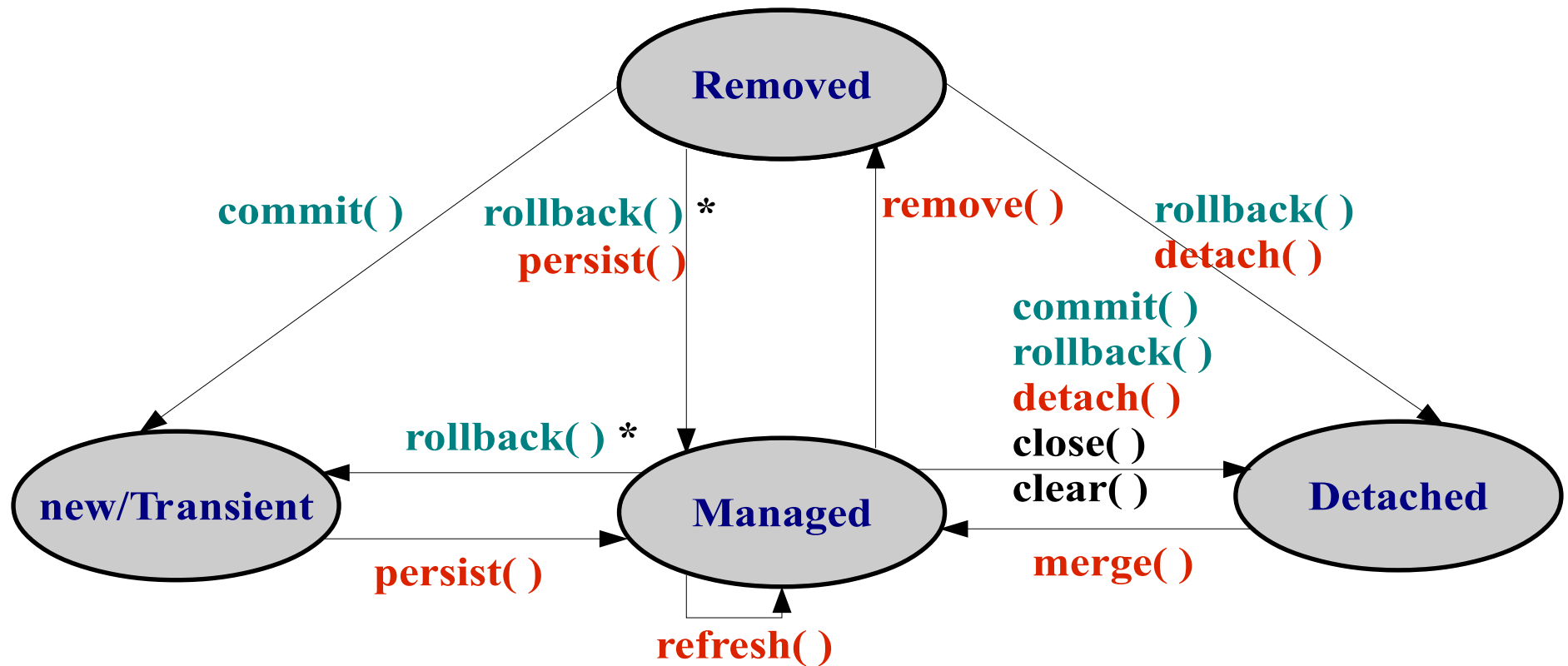
```
package javax.persistence;  
  
public interface EntityManagerFactory {  
    EntityManager createEntityManager();  
    EntityManager createEntityManager(Map map);  
    void close();  
    public boolean isOpen();  
}
```

```
package javax.persistence;

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface PersistenceContext {
    String name() default "";
    String unitName() default "";
    PersistenceProperty[] properties() default {};
    PersistenceContextType type() default /* type can be EXTENDED */
        PersistenceContextType.TRANSACTION; /* Default value */
}
```

```
@javax.ejb.Stateless
public class RentingAgentBean implements RentingAgentRemote {
    @PersistenceContext(unitName="carDatabase") ←
    private javax.persistence.EntityManager entityManager;
}

@javax.ejb.Stateful
public class RentingAgentBean implements RentingAgentRemote {
    @PersistenceContext(unitName="carDatabase", type=EXTENDED) ←
    private javax.persistence.EntityManager extendedEntityManager;
}
```



\* = Extended Persistence Context



```
persist(Object)
merge(T) : T
remove(Object)
refresh(Object)
refresh(Object, LockModeType)
lock(Object, LockModeType)
find(Class<T>, Object) : T
find(Class<T>, Object, LockModeType) : <T>
getReference(Class<T>, Object) : T
detach(Object)
getDelegate() : Object
contains(Object) : boolean
getCriteriaBuilder() : CriteriaBuilder
void flush()
void clear()
setFlushMode(FlushModeType)
getFlushMode() : FlushModeType
createQuery(String) : Query
createNamedQuery(String) : Query
createNativeQuery(String) : Query
createNativeQuery(String, Class) : Query
joinTransaction()
getTransaktion: EntityTransaction
close()
isOpen()
```

```
public void persist(Object entity);
```

```
Customer customer = new Customer();  
customer.setFirstName("Francis");  
entityManager.persist(customer);
```

- ▶ **persist( )** markiert eine neue Instanz als Managed Instanz
  - Nach **flush** oder **commit** wird die neue Instanz in die Datenbank geschrieben
  - **IllegalArgumentException** wird geworfen, falls die neue Instanz als **Detached** markiert ist
  - **TransactionRequiredException** wird geworfen, wenn diese Methode auf einen Transaction Scope Persistence Context aufgerufen wird

```
public void remove(Object entity);
```

```
@javax.ejb.TransactionAttribute(TransactionAttributeType.REQUIRED)
public void removeUser(long id) {
    User user = entityManager.find(User.class, id);
    entityManager.remove(user); }
```

- ▶ **remove ( )** markiert Managed Instanz als Removed Instanz
  - Nach flush oder commit wird die Managed Instanz aus der Datenbank gelöscht
  - **IllegalArgumentException** wird geworfen, falls diese Instanz als **Detached** markiert ist
  - **TransactionRequiredException** wird geworfen, wenn diese Methode auf einen Transaction Scope Persistence Context aufgerufen wird

```
@javax.ejb.TransactionAttribute(TransactionAttributeType.REQUIRED)
public void refreshUser(long id) {
    User user = entityManager.find(User.class, id);
    entityManager.refresh(user);

    // alternativ auch mit implizitem Lock moeglich
    entityManager.refresh(user, LockModeType.READ);
}
```

- ▼ **refresh( )** aktualisiert eine Managed Instanz mit dem neuen Zustand
  - **IllegalArgumentException** wird geworfen, falls die Instanz eine Detached Instanz ist
  - **TransactionRequiredException** wird geworfen, falls die Methode auf einen Transaction-Scoped PC aufgerufen wird
  - **EntityNotFoundException** wird geworfen, falls diese Instanz aus der Datenbank gelöscht wurde

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

```
@javax.ejb.TransactionAttribute(TransactionAttributeType  
.REQUIRED)  
public User updateUser(User detachedUser) {  
    User copyUser = entityManager.merge(detachedUser) ;  
    return copyUser ;  
}
```

- merge ( ) gibt eine Kopie einer Managed Instanz von einer gegebenen Detached Instanz zurück
  - IllegalArgumentException wird geworfen, falls eine Instanz als Removed Instanz markiert ist
  - TransactionRequiredException wird geworfen, falls die Methode auf einen Transaction Scoped Persistence Context aufgerufen wird

```
public void lock(Object entity, LockModeType lockMode);
```

- ▼ **lock ( )** sperrt eine gegebene Instanz durch einen bestimmten LockMode
- ▼ **javax.persistence.LockModeType**
  - **READ**
    - andere Transaktionen können das Objekt nur gleichzeitig lesen
  - **WRITE**
    - andere Transaktionen können nicht das Objekt gleichzeitig lesen oder schreiben

```
public CriteriaBuilder getCriteriaBuilder();
```

- ▼ Erzeugt ein Objekt zur programmatischen Erstellung von Abfragen
- ▼ Mehr dazu später (Criteria-API)

```
public void detachCustomer (int customerId){  
    Customer c = (Customer) em.find(Customer.class,  
                                     customerId) ;  
    if(c != null)  
        em.detach(c) ;  
}
```

- ▼ Entkoppelt eine Entity von ihrem Persistence-Context
- ▼ Änderungen an der Entity sind nicht mehr persistent
- ▼ Wiederaufnahme in den Persistence-Context über `merge`



```
public Query createQuery(String ejbqlString);  
  
public Query createNamedQuery(String name);  
  
public Query createNativeQuery(String sqlString);  
public Query createNativeQuery(String sqlString, Class resultClass);  
public Query createNativeQuery(String sqlString, String resultSetMapping);
```

```
Query query = entityManager.  
    createQuery("c from Customer c where id = 1");  
Customer cust = (Customer) query.getSingleResult();
```

```
public <T> T find(Class<T> entityClass, Object primaryKey);  
  
public <T> T find(Class<T> entityClass, Object primaryKey,  
                  LockModeType lockMode);
```

```
Customer customer = entityManager.find(Customer.class, 1);
```

- ▼ **find( )** gibt **null** zurück, falls die Instanz nicht in der Datenbank existiert
  - die Parameter von **find()** sind:
    - Klasse der Persistence-Entity
    - Primärschlüssel der Persistence-Entity
  - die Methode **find( )** benutzt Java Generics, um das Casting zu vermeiden
  - **IllegalArgumentException** für falsche Parameter

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

```
Customer customer = null;
try{
    customer = entityManager.getReference(Customer.class, 1);
}catch (EntityNotFoundException notFound) {
    //logische Besserung
}
```

▼ **getReference ( )** liefert eine Referenz auf die (noch nicht geladene) Instanz des Objekts, wirft aber

- **EntityNotFoundException** für nicht existierende Entities
- **IllegalArgumentException** für falsche Parameter.
- die Parameter sind
  - Klasse der Persistence-Entity
  - Primärschlüssel der Persistence-Entity

## contains ( )

- hat als Parameter die Instanz der Persistence-Entity und
- gibt **true** zurück, falls die Instanz vom Persistence Context verwaltet wird

```
Customer werner = new Customer("Eberling", "Werner");
em.persist(werner);

//The new Customer should be managed now
Assert.assertTrue("Customer should be managed now, after
    Call to persist.", em.contains(werner) );
```

```
//Setzt den Flush Modus für alle Objekte im PC
public void setFlushMode(FlushModeType flushMode);

//Gibt den Flush Modus für alle Objekte im PC
public FlushModeType getFlushMode();

//Löscht den Persistence Context und alle Managed
//Objekte werden detached Objekte
public void clear();
```

- Der FlushMode kontrolliert, ob die transaktionellen Änderungen vor der Ausführung von Abfragen mit der Datenbank synchronisiert wird
- javax.persistence.FlushModeType** hat zwei Konstanten
  - COMMIT** Synchronisation nur am Ende der Transaktion
  - AUTO** Synchronisation kann auch vor der Ausführung von Abfragen geschehen

```
public EntityTransaction getTransaction();
```

```
public interface EntityTransaction {  
    public void begin();  
    public void commit();  
    public void rollback();  
    public boolean isActive();  
}
```

- ▶ **EntityManager.getTransaction( )** wird in einer Nicht-Java EE Umgebung angewandt
  - **begin( )** wirft **IllegalStateException**, falls **EntityTransaction** schon aktiv ist
  - **commit( )** und **rollback( )** wirft **IllegalStateException**, falls eine Transaktion noch nicht aktiv ist

```
public void close();  
  
public boolean isOpen();
```

- ▼ Freigabe von allen Ressourcen einer EntityManager Instanz durch die Methode `close()`
  - Der Persistence Context wird beendet
  - Alle Managed Entities der EntityManager Instanz werden Detached Entities
  - Alle vorhandenen **Query** Instanzen werden ungültig
- ▼ `isOpen()` auf einer geschlossenen EntityManager Instanz wirft eine **IllegalStateException**

# Einfache Mappings



## ▼ Persistence Entities

- sind einfache Java Klassen
- werden wie ein normales Java Objekt alloziert und durch den Dienst EntityManager automatisch verwaltet
- benötigen einen Default-Konstruktor
- müssen der JavaBeans Konvention entsprechen und serialisierbar sein

## ▼ Java Persistence API erfordert nur zwei Annotationen von Meta-Informationen

- `@javax.persistence.Entity` bildet eine Klasse auf eine Datenbank ab
  - `@Entity` hat ein `name( )` Attribut, um die Entity in einer EJB QL zu referenzieren
- `@javax.persistence.Id` markiert eine Eigenschaft als Primärschlüssel

- ▼ alle andere Eigenschaften der Klasse werden auf Spalten der Datenbank mit dem gleichen Namen abgebildet
- ▼ XML Mapping kann anstatt Annotations benutzt werden
  - **orm.xml** in **META-INF/**
  - Deklarieren der Mapping Datei in ***persistence.xml*** (<mapping-file/>)

```
package de.mathema.domain;

@javax.persistence.Entity
public class Customer implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private Long id;
    private String lastName;
    private String firstName;

    @javax.persistence.Id
    public Long getId() {return id;}
    public void setId(Long id) {this.id = id;}

    public String getFirstName() {return this.firstName;}
    public void setFirstName(String firstName){
        this.firstName = firstName;}

    public String getLastName() {return this.lastName;}
    public void setLastName(String lastName) {
        this.lastName = lastName;}

    public String toString() {
        return "Customer#" + getId() + "(" + getLastName() + ")";
    }
}
```

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/
orm_1_0.xsd" version="1.0">
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <attributes>
      <id name="id"/>
    </attributes>
  </entity>
</entity-mappings>
```

- ▶ **<entity-mappings>** das Wurzelelement
  - **<entity>** definiert die Entity Klasse und den Typ des Zugriffs: **PROPERTY** oder **FIELD**
  - **<attributes>** hat **<id>** als Unterelement
    - **<id>** definiert ein Attribut als Primärschlüssel
- ▶ Der OR-Mapper nimmt alle anderen Eigenschaften der Klasse als persistente Eigenschaften an

## ▼ @javax.persistence.Table

- sagt dem EntityManager Dienst den Namen der Tabelle, auf den die Entity abgebildet wird
- diese Annotation wird oft in einer Bottom-Up Strategie angewandt
- **name()** Attribut legt den Namen der Tabelle einer Datenbank fest
- **schema()** und **catalog()** identifizieren den Katalog und das Schema in einer relationalen Datenbank

## ▼ @javax.persistence.Column

- beschreibt, wie eine bestimmte Eigenschaft auf eine Spalte in der Datenbank abgebildet wird
- **name()** Attribut legt den Namen der Spalte fest
- **length()** Attribut bestimmt die Länge einer Eigenschaft

```
package de.mathema.domain;

@javax.persistence.Entity
@javax.persistence.Table(name="CUSTOMER_TABLE")
public class Customer implements java.io.Serializable {
    @javax.persistence.Id
    @javax.persistence.Column(name="CUSTID",nullable=false,
        columnDefinition="integer")
    private long id;
    @javax.persistence.Column(name="LAST_NAME",length=50,nullable=false)
    private String lastName;
    @javax.persistence.Column(name="FIRST_NAME",length=50,nullable=false)
    private String firstName;

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

    public String getFirstName() {return firstName;}
    public void setFirstName(String firstName) {this.firstName = firstName;}

    public String getLastName() {return lastName;}
    public void setLastName(String lastName) {this.lastName = lastName;}
}
```

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/
orm_1_0.xsd" version="1.0">
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <table name="CUSTOMER_TABLE"/>
    <attributes>
      <id name="id">
        <column name="CUSTID"
          nullable="false"
          column-definition="integer"/>
      </id>
      <basic name="firstName">
        <column name="FIRST_NAME"
          nullable="false"
          lenght="50"/>
      </basic>
      ...
    </attributes>
  </entity>
</entity-mappings>
```

```
package javax.persistence;

@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
}
```

```
package javax.persistence;

@Target({}) @Retention(RUNTIME)
public @interface UniqueConstraint {
    String[] columnNames();
}
```



- ▼ @Column hat ein äquivalentes XML-Element **<column>**
- ▼ **Insertable( ) & updateable()**
  - spezifizieren, ob eine Spalte in **SQL INSERT** oder **UPDATE** beigefügt werden soll
- ▼ **<column>** Element ist ein Unterelement von Attributen:
  - **<id>**
  - **<basic>**
  - **<temporal>**
  - **<lob>**
  - **<enumerated>**

```
package javax.persistence;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {

    String name() default "";
    boolean unique()

        default false;
    boolean nullable()

        default true;
    boolean insertable()

        default true;
    boolean updatable()

        default true;
    String columnDefinition()

        default "";

    String table() default "";
    int length() default 255;
    int precision() default 0;
    int scale() default 0;
}
```

```
package javax.persistence;
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Id {}
```

```
package javax.persistence;
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface GeneratedValue {
    GenerationType strategy() default
        GenerationType.AUTO;
    String generator() default "";
}

package javax.persistence;
public enum GenerationType {
    TABLE, SEQUENCE, IDENTITY, AUTO
};
```

- ▶ **@javax.persistence.Id** identifiziert einen Primärschlüssel für eine Tabelle
- ▶ **@javax.persistence.GeneratedValue** für die automatische Generierung von Primärschlüsseln
  - Generierung erfordert einen bestimmten **GenerationType**
    - **AUTO** (Default Wert)
    - **IDENTITY** (**AUTO** kann durch **IDENTITY** ersetzt werden)
    - andere Typen mit zusätzlichen Meta-Informationen

```
package de.mathema.domain;

@javax.persistence.Entity
@javax.persistence.Table(name="CUSTOMER_TABLE")
public class Customer implements java.io.Serializable {
    private long id;
    ...
    @javax.persistence.Id
    @javax.persistence.GeneratedValue
    @javax.persistence.Column(name="CUSTID", nullable=false,
                              columnDefinition="integer")
    public long getId() {return id;}
    public void setId(long id) {this.id = id;}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <table name="CUSTOMER_TABLE"/>
    <attributes>
      <id name="id">
        <generated-value strategy="AUTO"/>
        ...
      </attributes>
    </entity>
  </entity-mappings>
```

```
package javax.persistence;
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface TableGenerator {
    String name();
    String table() default "";
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    UniqueConstraint[] uniqueConstraints() default {};
}
```

- ▶ ***name()***: benutzt Name von **@TableGenerator** und von **@Id**
- ▶ ***table()***: beschreibt die Definition der Tabelle
- ▶ ***pkColumnName()***: identifiziert den generierten Schlüssel
- ▶ ***valueColumnName()***: Name des Zählers für den generierten Schlüssel
- ▶ ***pkColumnValue()***: der Primärschlüssel
- ▶ ***allocationSize()***: Größe der Inkrementierung

```
# Die Strategie TABLE bestimmt eine relationale Tabelle für die
# Generierung von numerischen Schlüsseln
CREATE TABLE CUST_GENERATOR_TABLE
(
  PRIMARY_KEY_COLUMN VARCHAR not null,
  VALUE_COLUMN long not null
);
```

```
@javax.persistence.Entity
public class Customer implements java.io.Serializable {
    private long id;
    private String LastName;
    private String FirstName;
    @TableGenerator(name="CUST_GENERATOR", table="CUST_GENERATOR_TABLE",
        pkColumnName="PRIMARY_KEY_COLUMN",
        valueColumnName="VALUE_COLUMN",
        pkColumnValue="CUSTID",
        allocationSize=20)
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE,
        generator="CUST_GENERATOR")
    public long getId() {return id;}
    public void setId(long id) {this.id = id;}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <table-generator name="CUST_GENERATOR" table="CUST_GENERATOR_TABLE"
      pk-column-value="PRIMARY_KEY_COLUMN"
      value-column-name="VALUE_COLUMN"
      pk-column-value="CUSTID"
      allocation-size="20"/>
    <attributes>
      <id name="id">
        <generated-value strategy="TABLE" generator="CUST_GENERATOR"/>
      </id>
    </attributes>
  </entity>
</entity-mappings>
```

- **<table-generator>** ist ein Unterelement von **<entity>**
  - ihre Attribute sind genau so wie die der `@javax.persistence.TableGenerator` Annotation
  - nicht vergessen den Generator **<generated-value>** zu setzen!

```
package javax.persistence;  
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)  
public @interface SequenceGenerator {  
    String name();  
    String sequenceName() default "";  
    int initialValue() default 1;  
    int allocationSize() default 50;  
}
```

- ▼ **name()**: Name, der von dem **generator** Attribut des **@javax.persistence.GeneratedValue** benutzt wird
- ▼ **sequenceName()**: definiert die Sequence Tabelle, die aus der Datenbank benutzt wird

```
package de.mathema.domain;
import javax.persistence.*;

@Entity
public class Customer implements java.io.Serializable {
    private long id;
    private String LastName;
    private String FirstName;
    @SequenceGenerator(name="CUSTOMER_SEQUENCE",
        sequenceName="CUST_SEQ")

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
        generator="CUSTOMER_SEQUENCE")
    public long getId() {return id;}
    public void setId(long id) {this.id = id;}

    @Column(name="FIRST_NAME",length=50,nullable=false)
    public String getFirstName() {return FirstName;}
    public void setFirstName(String firstName) {FirstName = firstName;}

    @Column(name="LAST_NAME",length=50,nullable=false)
    public String getLastName() {return LastName;}
    public void setLastName(String lastName) {LastName = lastName;}
}
```



```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <sequence-generator name="CUSTOMER_SEQUENCE"
      sequence-name="CUST_SEQ"
      initial-value="0"/>
    <attributes>
      <id name="id">
        <generated-value strategy="SEQUENCE" generator="CUSTOMER_SEQUENCE"/>
      </id>
    </attributes>
  </entity>
</entity-mappings>
```

- **<sequence-generator>** ist ein Unterelement von **<entity>**
  - ihre Attribute sind genau so wie die der `@javax.persistence.SequenceGenerator` Annotation
  - nicht vergessen den Generator im Tag **<generated-value>** zu setzen

- ▼ EJB 2.1 Primärschlüssel Stil
- ▼ Java Persistence API stellt Annotations zur Verfügung:
  - `@javax.persistence.IdClass`
  - `@javax.persistence.EmbeddedId`
  - `@javax.persistence.Embeddable`
- ▼ automatische Generierung wird nicht von Composite Keys und Primary Key Classes unterstützt
- ▼ Klasse als Primärschlüssel muss diese Anforderungen erfüllen:
  - Klasse muss `java.io.Serializable` implementieren
  - Klasse muss einen No-Arg Konstruktor haben
  - Klasse muss die `equals()` und `hashCode()` Methode implementieren

```
package de.mathema.domain;

import javax.persistence.*;

@Entity
@IdClass(CustomerPK.class)
public class Customer implements
    java.io.Serializable {
    private String firstName;
    private String lastName;
    private long ssn;
    public String getFirstName()
    {return firstName;}
    public void setFirstName
    (String firstName)
    {this.firstName = firstName;}

    @Id
    public String getLastName()
    {return lastName;}
    public void setLastName
    (String lastName)
    {this.lastName = lastName;}

    @Id
    public long getSsn(){return ssn;}
    public void setSsn(long ssn)
    {this.ssn = ssn;}
}
```

```
package de.mathema.domain;
public class CustomerPK
    implements java.io.Serializable {
    private String lastName;
    private long ssn;

    ...

    public boolean equals(Object obj){
        if (obj==this) return true;
        if (!(obj instanceof CustomerPK))
            return false;
        CustomerPK pk = (CustomerPK)obj;
        if (!lastName.equals(pk.lastName))
            return false;
        if (ssn != pk.ssn) return false;
        return true;
    }

    public int hashCode( ){
        return lastName.hashCode()
            +(int)ssn;
    }
}
```

```
<entity-mappings>
  <entity class="com.titan.domain.Customer"
    access="PROPERTY">
    <id-class>de.mathema.domain.CustomerPK</id-class>
    <attributes>
      <id name="lastName" />
      <id name="ssn" />
    </attributes>
  </entity>
</entity-mappings>
```

```
package de.mathema.domain;

import javax.persistence.*;

@Entity
public class Customer
    implements
        java.io.Serializable {
    private String firstName;
    private CustomerPK pk;

    public String getFirstName()
    {return firstName;}
    public void setFirstName
        (String firstName)
    {this.firstName=firstName;}

    @EmbeddedId
    public CustomerPK getPk()
    {return pk;}
    public void setPk(CustomerPK pk)
    {this.pk=pk;}
}
```

```
package de.mathema.domain;
import javax.persistence.Column;

@Embeddable
public class CustomerPK
    implements java.io.Serializable {
    private String lastName;
    private long ssn;

    @Column(name="CUSTOMER_LAST_NAME")
    public String getLastName()
    {return this.lastName;}
    public void setLastName(String lastName)
    {this.lastName=lastName;}

    ...
    public boolean equals(Object obj){
        if(obj == this) return true;
        if(!(obj instanceof CustomerPK))
            return false;
        CustomerPK pk = (CustomerPK)obj;
        if(!lastName.equals(pk.lastName))
            return false;
        if(ssn != pk.ssn) return false;
        return true;
    }
    public int hashCode() {
        return lastName.hashCode()
            +(int) ssn;
    }
}
```

```
<entity-mappings>
<embeddable class="de.mathema.domain.CustomerPK"
  access-type="PROPERTY">
  <embeddable-attributes>
    <basic name="lastName">
      <column name="CUSTOMER_LAST_NAME"/>
    </basic>
    <basic name="ssn">
      <column name="CUSTOMER_SSN"/>
    </basic>
  </embeddable-attributes>
</embeddable>
<entity class="de.mathema.domain.Customer" access="PROPERTY">
  <attributes>
    <embedded-id name="pk">
      <attribute-override name="lastName">
        <column name="LAST_NAME"/>
      </attribute-override>
      <attribute-override name="ssn">
        <column name="SSN"/>
      </attribute-override>
    </embedded-id>
  </attributes>
</entity>
</entity-mappings>
```

```
package de.mathema.domain;
import javax.persistence.*;
@Entity
public class Customer
    implements java.io.Serializable {
    private long id;
    private Address address;
    ...
    @Embedded
    public Address getAddress() {
        return address;
    }
}
```

```
package de.mathema.domain;

import javax.persistence.Embeddable;

@Embeddable
public class Address
    extends Serializable{
    private String street;
    private String city;
    private String state;

    ...
}
```

- Java Persistence API ermöglicht, in Persistence Entities eingebettete Java Objekte zu benutzen
- Eingebettete Java Objekte (dependent objects)
  - haben keine Identität
  - werden mit `@javax.persistence.Embedded` markiert

```
<entity-mappings>
<embeddable class="de.mathema.domain.Address"
  access-type="PROPERTY"/>
<entity class="de.mathema.domain.Customer"
  access="PROPERTY">
  <attributes>
    <id name="id"/>
    <embedded name="address">
      <attribute-override name="street">
        <column name="CUST_STREET"/>
      </attribute-override>
      <attribute-override name="city">
        <column name="CUST_CITY"/>
      </attribute-override>
      <attribute-override name="state">
        <column name="CUST_STATE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
</entity-mappings>
```



```
@Entity
public class Customer
    implements java.io.Serializable {
    private long id;
    @Embedded Address address;
    ...
}
```

```
@Embeddable
public class Address {

    @Embedded Street street
}
```

```
@Embeddable
public class Street {

    String streetName;
    String hsNr;
}
```

- ▼ Embeddables können auch in Collections verwaltet werden
  - Speicherung erfolgt in eigener Tabelle

```
@Entity
public class Customer {

    @ElementCollection
    Set<Phone> phoneNumber = new HashSet<Phone>();
    ...
}
```

```
@Embeddable
public class Phone {

    String countryCode;
    String regionalCode;
    String callNumber;

}
```

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface CollectionTable {
    String name() default "";
    String catalog() default "";
    JoinColumn[] joinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

## ▼ **@javax.persistence.Transient**

- der Persistence Manager ignoriert alle Eigenschaften, die mit **@javax.persistence.Transient** markiert sind

## ▼ **@javax.persistence.Basic** und FetchType (LAZY, EAGER)

- **@javax.persistence.Basic**
  - markiert alle Java primitive oder Java Wrapper Typen
  - hat Attribute *fetch()* und *optional()*

- ▼ **@Temporal** und `TemporalType(DATE, TIME, TIMESTAMP)`
  - verwendbar für `java.util.Date` oder `java.util.Calendar`
- ▼ **@javax.persistence.Lob** ermöglicht die Markierung von primitiven Typen als `java.sql.Blob` oder `java.sql.Clob`
  - **Blob** für Java Typen *byte[ ]*, *Byte[ ]* oder *java.io.Serializable*
  - **Clob** für Java Typen *char[ ]*, *Character[ ]* oder *java.lang.String*

## ▼ **@javax.persistence.Enumerated**

- wird verwendet, um Java enum Typen zu markieren

## ▼ **@javax.persistence.Version**

- wird für die Optimistic Concurrency benutzt

## ▼ **@javax.persistence.SecondaryTable**

- Java Persistence API ermöglicht durch **@SecondaryTable** die Abbildung einer Persistence Entity auf eine oder mehrere Tabelle(n)

## ▼ **@javax.persistence.Access**

- definiert die Zugriffsvariante für eine Attribut der Entität

## ▼ **Constraint-Angaben bisher nur für DDL-Skripte**

- `@Column(nullable=false, length=256)`
- `@Basic(optional=false)`
- ...
- Keine Berücksichtigung durch den Persistence-Provider
- Attributwerte müssen nicht mit diesen Angaben übereinstimmen!

- ▼ Eigener Standard innerhalb der JEE
- ▼ Framework zur Validierung von Bean-Attributen
  - Vordefinierte Menge von Standard-Constraints
  - Möglichkeit zur Implementierung eigener Constraints
  - In den Persistence-Provider integriert
- ▼ Prüfung der Constraints
  - Vor der Persistierung
  - Vor einem Update
  - Vor der Löschung (optional)



## ▼ Vordefinierte Constraints in `javax.validation.constraints`

```
@Entity
public class Kunde {
    @NotNull
    @Size(min=2,max=25)
    String name;

    @NotNull
    @Size(min=2,max=25)
    String vorname;

    @Embedded
    @Valid
    Adresse adresse;
}
```

## Implementierung eigener Constraints möglich

```
@Constraint(validatedBy = KundenNummerValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface KundenNummer {
    String message() default "{de.mathema.schulung.KundenNummer.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

public class KundenNummerValidator implements
    ConstraintValidator<KundenNummer, String> {

    //...
}
```

# Beziehungen

- ▼ die unidirektionale **One-to-one** Beziehung
  - ein Kunde hat eine unidirektionale Beziehung mit der Adresse
- ▼ die bidirektionale **One-to-one** Beziehung
  - ein Kunde hat eine bidirektionale Beziehung mit der Kreditkarte
- ▼ die unidirektionale **One-to-many** Beziehung
  - ein Kunde hat eine unidirektionale Beziehung mit dem Telefon
- ▼ die bidirektionale **One-to-many / Many-to-One** Beziehung
  - ein Anbieter hat eine bidirektionale Beziehung mit dem Auto
- ▼ die bidirektionale **Many-to-many** Beziehung
  - eine Reservierung kann viele Fahrer haben und jeder Fahrer kann für viele Reservierungen eingetragen sein

- ▼ **@OneToOne** beschreibt eine Beziehung, die
  - mit **@JoinColumn** auf einen Fremdschlüssel,
  - **@JoinColumns** auf einen zusammengesetzter Primärschlüssel oder
  - **@PrimaryKeyJoinColumn** auf Primärschlüssel von beiden Persistence-Entities abgebildet wird



```
@javax.persistence.Entity
public class Customer implements java.io.Serializable{
    ...
    private Address address;
    ...
    @javax.persistence.OneToOne(cascade={CascadeType.ALL})
    @javax.persistence.JoinColumn(name="ADDRESSID")
    public Address getAddress() {return this.address;}
    public void setAddress(Address address){this.address = address;}
}
```

```
package javax.persistence;
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
}
```

Attribut	Typ	Bedeutung	Default
targetEntity	Class	Entitätsklasse, die Ziel der Assoziation ist; nur erforderlich, wenn dies nicht aus dem Attributstyp hervorgeht (optional)	Typ des Attributs
cascade	CascadeType[]	Angabe der Operationen, die an das Ziel der Assoziation weitergeleitet werden müssen (optional, siehe Beispiel für unidirektionale Beziehung)	-
fetch	FetchType	bestimmt das Ladeverhalten des Persistence-Providers in Bezug auf die Beziehung (optional)	EAGER
optional	Boolean	gibt an, ob die Assoziation belegt sein muss oder nicht (optional)	true
mappedBy	String	darf an nur einer Seite der Beziehung – der nicht-führenden – angegeben werden	

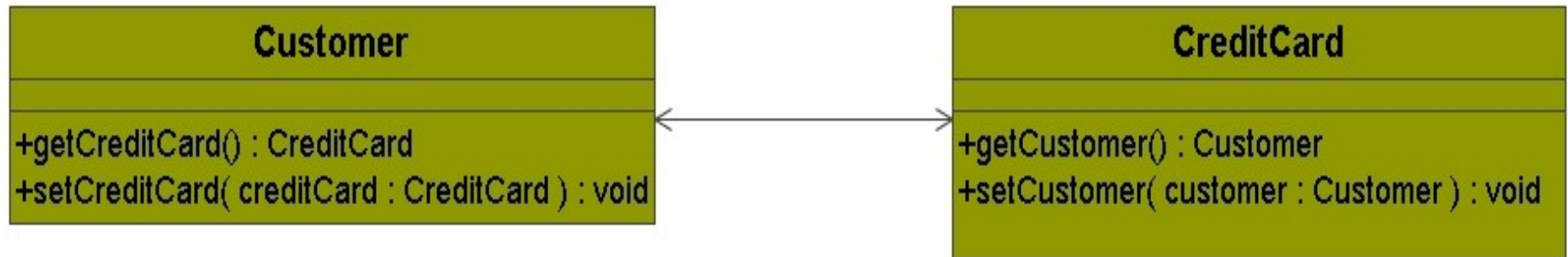
```
package javax.persistence;

public @interface JoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer" access="PROPERTY">
    <attributes>
      <id name="id"><generated-value/></id>
      <one-to-one name="address"
        targetEntity="de.mathema.domain.Address"
        fetch="LAZY"
        optional="true">
        <cascade>ALL</cascade>
        <join-column name="ADDRESSID"/>
      </one-to-one>
    </attributes>
  </entity></entity-mappings>
```



- Für eine bidirektionale Beziehung wird
  - `@javax.persistence.OneToOne` in dem referenzierten Objekt mit dem Attribut ***mappedBy()*** angewandt



```
package de.mathema.domain;
@javax.persistence.Entity
public class Customer implements
    java.io.Serializable{
    private CreditCard creditCard;

    ...

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="CREDITCARDID")
    public CreditCard getCreditCard()
    {return creditCard;}

    // Sicherstellung der Multiplizität
    // und Navigierbarkeit
    public void setCreditCard
        (CreditCard creditCard)
    {this.creditCard = creditCard;}
}
```

```
package de.mathema.domain;
@javax.persistence.Entity
public class CreditCard implements
    java.io.Serializable{
    private Customer customer;

    ...

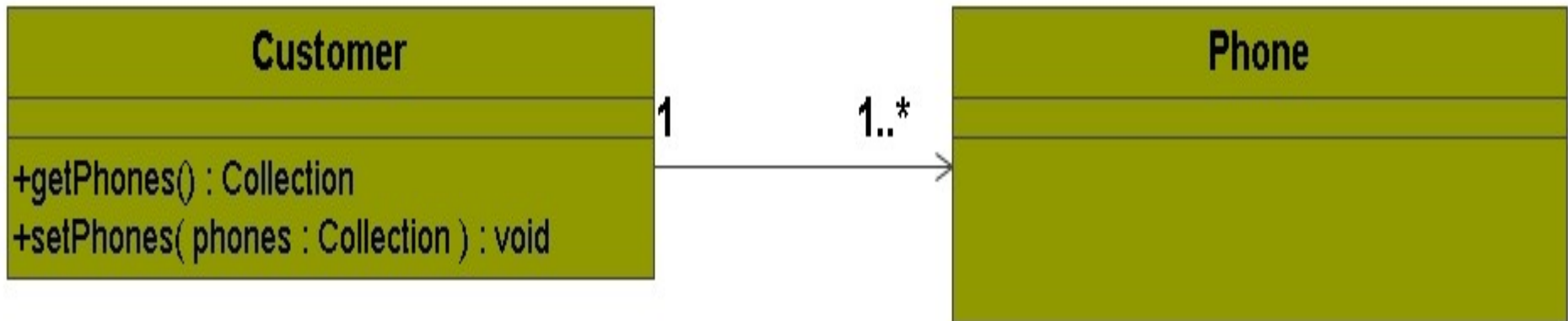
    @OneToOne(mappedBy="creditCard")
    public Customer getCustomer()
    {return customer;}

    // Sicherstellung der Multiplizität
    // und Navigierbarkeit
    public void setCustomer
        (Customer customer)
    {this.customer = customer;}

}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer"
    access="PROPERTY">
    <attributes>
      <id name="id"><generated-value/></id>
      <one-to-one name="creditCard"
        target-entity="de.mathema.domain.CreditCard"
        fetch="LAZY">
        <cascade-all/>
        <join-column name="CREDITCARDID"/>
      </one-to-one>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.CreditCard"
    access="PROPERTY">
    <attributes>
      <id name="id"><generated-value/></id>
      <one-to-one name="customer"
        target-entity="de.mathema.domain.Customer"
        mapped-by="creditCard"/>
    </attributes>
  </entity>
</entity-mappings>
```

- `@javax.persistence.OneToOne` wird für die Deklaration einer One-to-many Beziehung angewandt und benutzt
  - die Annotation `@javax.persistence.JoinColumn` oder
  - die Annotation `@javax.persistence.JoinTable` für die Abbildung auf Fremdschlüssel



```
package javax.persistence;
public @interface OneToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

```
package javax.persistence;
public @interface JoinTable {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    JoinColumn[] inverseJoinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

*Primärschlüssel eigener Seite*

*Primärschlüssel anderer Seite*

Attribut	Typ	Bedeutung	Default
targetEntity	Class	wie bei @OneToOne, hier jedoch etwas relevanter für den Fall der Anwendung untypisierter Kollektionen (optional)	Inhaltstyp der Collection, bei Anwendung von Generics
cascade	CascadeType[]	Angabe der Operationen, die an das Ziel der Assoziation weitergeleitet werden müssen (optional, siehe Beispiel für unidirektionale Beziehung)	-
fetch	FetchType	bestimmt das Ladeverhalten des Persistence-Providers in Bezug auf die Beziehung (optional)	LAZY
mappedBy	String	Pflichtattribut im Gegensatz zu @OneToOne	

```
package de.mathema.domain;
@javax.persistence.Entity
public class Phone implements java.io.Serializable {
    private long id;
    ...
    @javax.persistence.Column(name="NUMBER")
    public String getNumber() {return number;}
    public void setNumber(String nummer) {this.number = nummer;}
    ...
}
```

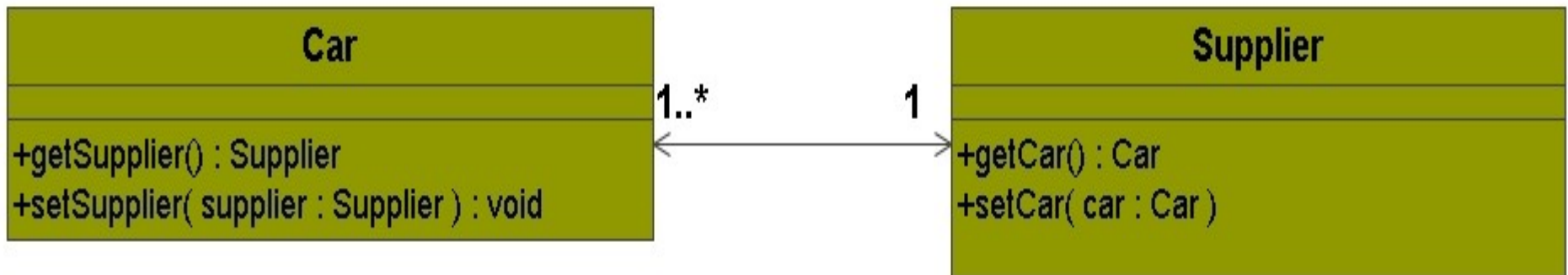
```
package de.mathema.domain;
@javax.persistence.Entity
public class Customer implements Serializable {
    private Address address;
    private Collection<Phone> phones;
    ...
    @javax.persistence.OneToMany(cascade={CascadeType.ALL})
    @JoinTable(name="CUSTOMER_PHONE",
        joinColumns={@JoinColumn(name="CUSTOMERID")},
        inverseJoinColumns={@JoinColumn(name="PHONEID")})
    public Collection<Phone> getPhones() {return phones;}
    public void setPhones(Collection<Phone> phones) {this.phones = phones;}
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer"
    access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <one-to-many name="phones"
        targetEntity="de.mathema.domain.Phone">
        <cascade-all />
        <join-table name="CUSTOMER_PHONE">
          <join-column name="CUSTOMERID" />
          <inverse-join-column name="PHONEID" />
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
</entity-mappings>
```



## ▼ die Beschreibung dieser Beziehung geschieht mit Hilfe von Annotations:

- `@javax.persistence.ManyToOne`
  - `@javax.persistence.JoinColumn` wird für die Abbildung auf einen Fremdschlüssel verwendet
- `@javax.persistence.OneToOne`
  - das Attribut ***mappedBy()*** wird nur für bidirektionale Beziehungen verwendet



```
package javax.persistence;

public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

- ▼ die Attributdefinition von `@javax.persistence.ManyToOne` ähnelt der von `@javax.persistence.OneToOne`

Attribut	Typ	Bedeutung	Default
targetEntity	Class	Entitätsklasse, die Ziel der Assoziation ist; nur erforderlich, wenn dies nicht aus dem Attributstyp hervorgeht (optional)	Typ des Attributs
cascade	CascadeType[]	Angabe der Operationen, die an das Ziel der Assoziation weitergeleitet werden müssen (optional, siehe Beispiel für unidirektionale Beziehung)	-
fetch	FetchType	bestimmt das Ladeverhalten des Persistence-Providers in Bezug auf die Beziehung (optional)	EAGER
optional	Boolean	gibt an, ob die Assoziation belegt sein muss oder nicht (optional)	true
mappedBy	String	darf an nur einer Seite der Beziehung – der nicht-führenden – angegeben werden	

```
package de.mathema.domain;
@javax.persistence.Entity
public class Car implements Serializable {
    ...
    private Supplier supplier;
    ....
    @javax.persistence.ManyToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="SUPPLIERID")
    public Supplier getSupplier() {return supplier;}
    public void setSupplier(Supplier supplier) {this.supplier = supplier;}
}
```

```
package de.mathema.domain;

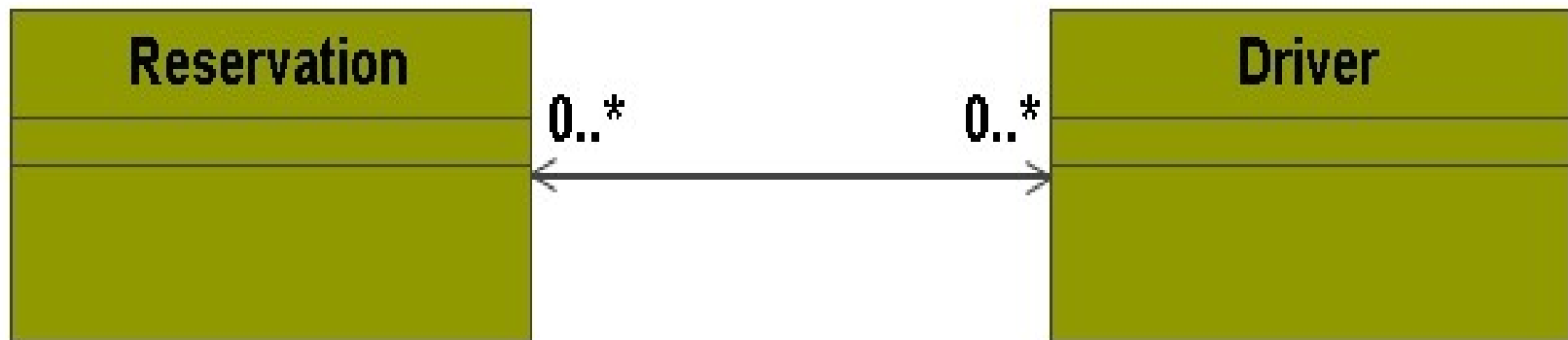
@javax.persistence.Entity
public class Supplier implements Serializable{
    private Set<Car> cars = new HashSet<Car>();
    ...
    @javax.persistence.OneToMany(mappedBy="supplier")
    public Set<Car> getCars() {return cars;}
    public void setCars(Set<Car> cars) {this.cars = cars;}
}
```

# Die bidirektionale One-to-many / Many-to-One Beziehung: XML Beispiel

85

```
<entity-mappings>
  <entity class="de.mathema.domain.Supplier" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <one-to-many name="car"
        target-entity="de.mathema.domain.Car" fetch="LAZY"
        mapped-by="supplier">
      </one-to-many>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Car" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <many-to-one name="supplier"
        target-entity="de.mathema.domain.Supplier"
        fetch="EAGER">
        <join-column name="SUPPLIERID" />
      </many-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

- ▼ Die Many-to-many Beziehung wird durch die `@javax.persistence.ManyToMany` definiert
- ▼ `@javax.persistence.JoinTable` definiert die Tabelle (Link Tabelle), die eine große Rolle in der Beziehung spielt.



```
package javax.persistence;
public @interface ManyToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

```
package de.mathema.domain;
@javax.persistence.Entity
public class Reservation implements Serializable {
    ...
    private Set<Driver> drivers = new HashSet<Driver>();
    // Sicherstellung der Multiplizität und Navigierbarkeit
    @javax.persistence.ManyToMany
    @javax.persistence.JoinTable(name="RESERVATION_DRIVER",
        joinColumns={@JoinColumn(name="RESERVATION_ID")},
        inverseJoinColumns={@JoinColumn(name="DRIVER_ID")})
    public Set<Driver> getDrivers() {return drivers;}
    public void setDrivers(Set<Driver> drivers) {this.drivers = drivers;}
}
```

```
package de.mathema.domain;
@javax.persistence.Entity
public class Driver implements Serializable{
    ...
    private Collection<Reservation> reservations = new ArrayList<Reservation>();
    // Sicherstellung der Multiplizität und Navigierbarkeit
    @javax.persistence.ManyToMany(mappedBy="drivers")
    public Collection<Reservation> getReservations() {return reservations;}
    public void setReservations(Collection<Reservation> reservations)
    {this.reservations = reservations;}
    ...
}
```



```
<entity-mappings>
  <entity class="de.mathema.domain.Reservation" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <many-to-many name="drivers"
        target-entity="de.mathema.domain.Driver" fetch="LAZY">
        <join-table name="RESERVATION_DRIVER">
          <join-column name="RESERVATION_ID" />
          <inverse-join-column name="DRIVER_ID" />
        </join-table>
      </many-to-many>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Driver" access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <many-to-many name="reservations"
        target-entity="de.mathema.domain.Reservation" fetch="LAZY"
        mapped-by="drivers">
      </many-to-many>
    </attributes>
  </entity>
</entity-mappings>
```

- Das automatische Schreiben/Lesen/Löschen/Aktualisieren von Beziehungen einer persistenten Instanz in die Datenbank
- setze das Attribut `cascade()` mit einem der Typen aus `javax.persistence.CascadeType` in den Beziehungen `@OneToOne`, `@OneToMany`, `@ManyToOne` und `@ManyToMany`

```
package javax.persistence;
public enum CascadeType {

    ALL, //ist eine Kombination aus allen Richtlinien

    PERSIST, //kaskadiert entityManager.persist() Operationen

    MERGE, //kaskadiert entityManager.merge() Operationen

    REMOVE, //kaskadiert entityManager.remove() Operationen

    REFRESH //kaskadiert entityManager.refresh() Operationen
}
```

```
package de.mathema.domain;

@javax.persistence.Entity
public class Customer implements Serializable{
    ...
    private Address address;
    ...
    @javax.persistence.OneToOne
        (cascade={CascadeType.PERSIST,CascadeType.REMOVE})
    @javax.persistence.JoinColumn(name="ADDRESS_ID")
    public Address getAddress() {return this.address;}
    public void setAddress(Address address)
        {this.address = address;}
    ...
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.Customer"
    access="PROPERTY">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
      <one-to-one name="address"
        targetEntity=
          "de.mathema.domain.Address" fetch="LAZY"
        optional="true">
        <cascade-persist />
        <cascade-remove />
        <primary-key-join-column />
      </one-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

## ▼ LAZY und EAGER Fetching

- durch ***fetch()*** in allen Objektbeziehungen

## ▼ LAZY Fetching

- das annotierte Attribut kann bei Bedarf nachgeladen werden

## ▼ EAGER Fetching

- alle Attribute werden sofort geladen

## ▼ LAZY Fetching in Verbindung mit FETCH JOIN Operation

- verbessert die Manipulation von Managed Objekten

# Vererbung

- ▼ EJB 3.1 Persistence API unterstützt drei Strategien für die Vererbungsabbildung
  - Single Table Per Class Hierarchy (default)
  - Table Per Concrete Class
  - Table Per Subclass
- ▼ Java Persistence API unterstützt zusätzlich
  - Implizite Polymorphie
  - Polymorphe Abfrage

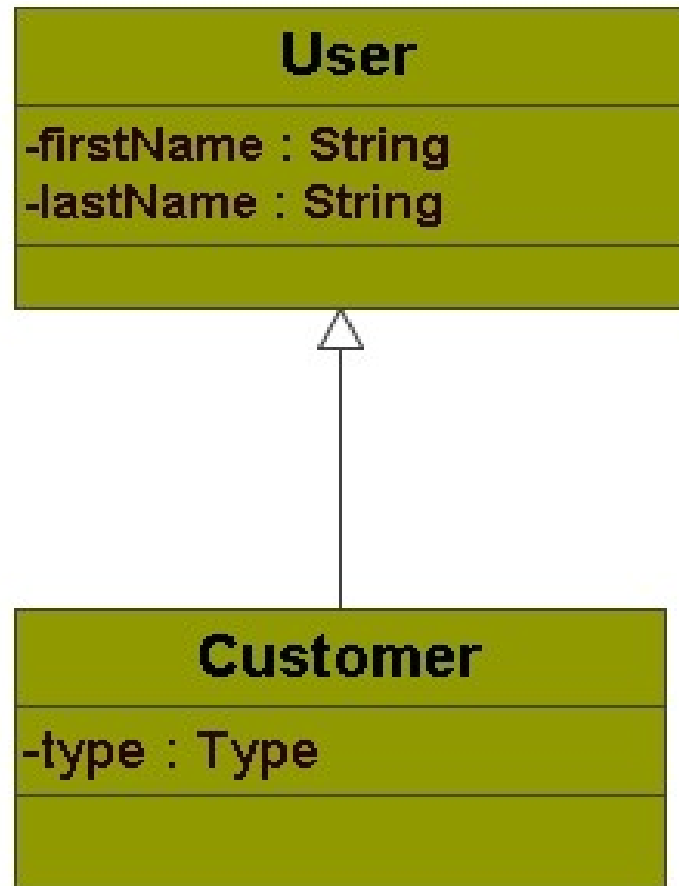
```
package javax.persistence;
public @interface Inheritance
    InheritanceType strategy()
default SINGLE_TABLE;
}
```

```
package javax.persistence;
public enum InheritanceType
{
    SINGLE_TABLE,
    TABLE_PER_CLASS,
    JOINED
};
```

```
package javax.persistence;
public @interface DiscriminatorColumn
{
    String name() default "DTYPE";
    DiscriminatorType
discriminatorType() default STRING;
    String columnDefinition()
default "";
    int length() default 31;
}
```

```
package javax.persistence;
public @interface
    DiscriminatorValue {
    String value();
}
```





USER
<b>id integer primary key not null, firstName varchar(255), lastName varchar(255), type varchar(255), DISCRIMINATOR varchar(31) not null</b>

- ▼ Einzige Tabelle für die gesamte Klassenhierarchie
  - erfordert eine zusätzliche Discriminator-Spalte
- ▼ Vorteile
  - Performanz
  - Einfachheit
- ▼ Nachteile
  - Spalten für die Attribute einer Unterklasse können den NULL-Wert annehmen

```
package de.mathema.domain;

@Entity
@Table(name="USERS")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIMINATOR",
    discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("USER")
public class User implements Serializable{
    ...
}
```

```
package de.mathema.domain;

@Entity
@DiscriminatorValue("CUST")
public class Customer extends User implements Serializable{

}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.User">
    <inheritance strategy="SINGLE_TABLE" />
    <discriminator-column name="DISCRIMINATOR"
      discriminator-type="STRING" />
    <discriminator-value>USER</discriminator-value>
    <attributes>
      <id>
        <generated-value />
      </id>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Customer">
    <discriminator-value>CUST</discriminator-value>
  </entity>
</entity-mappings>
```

User
<b>id integer primary key not null,</b> <b>firstName varchar(255),</b> <b>lastName varchar(255),</b> ...

Customer
<b>id integer primary key not null,</b> <b>firstName varchar(255),</b> <b>lastName varchar(255),</b> <b>type varchar(255),</b> ...

▼ Diese Strategie ist nicht zu empfehlen:

- Eine Tabelle für jede reale Unterklasse
- Polymorphie wird nicht unterstützt
- Evolution des Schemas wird immer komplexer

```
package de.mathema.domain;

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class User implements Serializable{
    ...
}
```

```
package de.mathema.domain;

@Entity
public class Customer extends User implements Serializable{
    ...
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.User">
    <inheritance strategy="TABLE_PER_CLASS"/>
    <attributes>
      <id>
        <generated-value/>
      </id>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Customer"/>
</entity-mappings>
```

- ▼ Alle Unterklassen besitzen ihre eigene Tabelle
- ▼ Gemeinsame Benutzung von gleichen Primärschlüsseln
- ▼ Vorteile
  - Komplette Normalisierung
  - Datenintegrität und Evolution des Schemas sind einfach
- ▼ Nachteil
  - Weniger performant als Table Per Class Hierarchy

User
<b>id integer primary key not null,</b> <b>firstName varchar(255),</b> <b>lastName varchar(255),</b> ...

Customer
<b>id integer primary key not null,</b> <b>type varchar(255),</b> ...



```
package de.mathema.domain;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class User implements Serializable{
    ...
}
```

```
package de.mathema.domain;

@Entity
public class Customer extends User implements Serializable{
    ...
}
```

```
<entity-mappings>
  <entity class="de.mathema.domain.User">
    <inheritance strategy="JOINED" />
    <attributes>
      <id>
        <generated-value/>
      </id>
    </attributes>
  </entity>
  <entity class="de.mathema.domain.Customer" />
</entity-mappings>
```

# Abfragen

## Java Persistence Query API

- völlig ausgearbeitetes Java Interface
- `javax.persistence.Query` wird zur Laufzeit über `javax.persistence.EntityManager` generiert

```
package javax.persistence;

public interface EntityManager {
    public Query createQuery(String jpqlString);
    public Query createNamedQuery(String name);
    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString,
        Class resultClass);
    public Query createNativeQuery(String sqlString,
        String resultSetMapping);
}
```

- ▼ String-Basiert
- ▼ Dynamisch und statisch möglich
- ▼ ABER
  - Keine Typsicherheit
  - Prüfung erst zur Laufzeit
- ▼ Typischere Queries über CriteriaBuilder
  - Benötigt Metamodell der Entitäten
  - Erstellt per AnnotationProcessor (statisch)
  - Abfrage am EntityManager (dynamisch)

```
package javax.persistence;

public interface Query {
    public List getResultList();
    public Object getSingleResult();
    public int executeUpdate();
    public Query setMaxResults(int maxResult);
    public Query setFirstResult(int startPosition);
    public Query setHint(String hintName, Object value);
    public Query setParameter(String name, Object value);
    public Query setParameter(String name,
        Date value, TemporalType temporalType);
    public Query setParameter(String name, Calendar value,
        TemporalType temporalType);
    public Query setParameter(int position, Object value);
    public Query setParameter(int position, Date value,
        TemporalType temporalType);
    public Query setParameter(int position, Calendar value,
        TemporalType temporalType);
    public Query setFlushMode(FlushModeType flushMode);
}
```

```
try{
Query query = entityManager.createQuery(
    "select c from Customer c where c.firstName='" + firstName +
    "' and c.lastName='" + lastName + "'");
    customer = (Customer)query.getSingleResult();
}catch(EntityNotFoundException notFound){
}catch(NonUniqueResultException nonUnique){
}
```

```
Query query = entityManager.createQuery(
    "select c from Customer c where c.firstName='" + firstname +
    "' and c.lastName='" + lastName + "'");
java.util.List<Customer> customers = query.getResultList();
```

## ▼ Query API unterstützt

- benannte Parameter(:name)
- JDBC-artige Parameter

```
...
Query query = manager.createQuery
("from Customer c where " +
"c.firstName=:firstName");

query.setParameter
(firstName, firstName);
...
```

```
...
Query query = manager.createQuery
("from Customer c where " +
"c.firstName=?1");
query.setParameter(1, firstName);
....
```



## ▼ Begrenzung auf das ResultSet

- Query API hat zwei eingebaute Methoden für dieses Szenario
- Query.setMaxResult( )
- Query.setFirstResult( )

```
public List getCustomer(int max, int index) {  
    Query query = manager.createQuery("from Customer cust");  
    return query.setMaxResults(max) .  
        setFirstResult(index) .  
        getResultList();  
}
```

## ▼ JPQL

- ist vollständig objektorientiert
- unterstützt Vererbung, Polymorphie und Assoziationen

## ▼ JPQL Features

- FROM, SELECT, WHERE, HAVING, ORDER BY, GROUP BY, JOIN Operationen,
- Aggregationsfunktionen
- Polymorphe Abfrage
- Projektion
- Dynamische Abfragen und benannte Parameter
- Subqueries
- Bulk Update
- Delete Update

## ▼ Abfrage sind case-insensitive

- **SeLeCT = sELEct = SELECT**
- *SELECT cust.lastName FROM Customer cust*
- ausgenommen sind Java Klassen Namen und Eingeschaften
  - **de.mathema.domain.CUSTOMER != de.mathema.domain.Customer**

## ▼ FROM-Klausel

- alle Instanzen vom Typ Customer
  - *FROM de.mathema.domain.Customer*
  - Alias "AS" kann in der Abfrage benutzt werden
    - *FROM Customer AS cust*

## ▼ Projektion

- *SELECT c.lastName, cc.number FROM Customer c, CreditCard cc WHERE c.creditCard = cc*

## ▼ IN Operator

- *SELECT d FROM Reservation As r, **IN**( r.drivers ) d*

## ▼ LEFT JOIN

- *SELECT c.firstName, c.lastName, p.number FROM Customer c **LEFT JOIN** c.phones p*

## ▼ FETCH JOIN

- *SELECT c FROM Customer c **LEFT JOIN FETCH** c.phones*

## ▼ DISTINCT

- *SELECT **DISTINCT** c FROM Reservation AS res, **IN** (res.drivers) cust*

## ▼ Aggregationsfunktionen

- AVG( ), SUM( ), MIN( ), MAX( ), COUNT(\*), COUNT( ), ...
  - *SELECT COUNT(cust) FROM Customer AS cust*

## ▼ Polymorphe Abfrage

- Abfrage auf eine Basisklasse gibt auch die Unterklassen zurück
- *FROM User user*

## ▼ WHERE-Klausel

- *FROM Customer cust WHERE cust.id = 1*

## ▼ ORDER BY-Klausel

- *FROM Customer cust ORDER BY cust.firstName ASC*

## ▼ GROUP BY-Klausel

- *SELECT c.lastName, COUNT(c) FROM Customer c GROUP BY c.id*

## ▼ Bulk UPDATE

- *UPDATE Reservation res SET res.car.price = (res.car.price + 6) WHERE EXISTS ( SELECT cust FROM res.customers cust WHERE cust.firstName = 'Serge' AND cust.lastName='Pagop' )*

## ▼ Bulk DELETE

- *DELETE Reservation res WHERE EXISTS ( SELECT cust WHERE cust.firstName = 'Francis' AND cust.lastName='Pouatcha' )*
- Vor der Ausführung von Bulk Operationen empfiehlt die Java Persistence API, die Methoden *EntityManager.flush()* und *EntityManager.clear()* auszuführen

## ▼ Subqueries

- *SELECT s FROM Supplier s WHERE (SELECT COUNT(c) FROM s.cars c) > 10*

- Formulierung einer nativen Abfrage mittels `EntityManager.createNativeQuery( )`

```
Query query = entityManager.createNativeQuery  
(  
    "SELECT p.id, p.number, p.type  
      FROM PHONE AS p",  
    Phone.class  
);
```

```
package javax.persistence;
public @interface NamedQueries {
    NamedQuery [] value ();
}
```

```
package javax.persistence;
public @interface NamedQuery {
    String name();
    String query();
    QueryHint[] hints()
    default {};
}
```

```
package de.mathema.domain;
@NamedQueries({
    @NamedQuery(name="getCustomer",
        query="SELECT c FROM Customer AS c WHERE c.lastName=:lastname")
})
@Entity
public class Customer extends User implements Serializable{
    ...
}
// in der SLSB
Query query =
    entityManager.createNamedQuery("getCustomer") .
    setParameter("lastName", lastName);
```



```
<entity-mappings>
  <named-query name="getCustomer">
    <query>
      SELECT c FROM Customer
      AS c WHERE c = :lastName
    </query>
  </named-query>
</entity-mappings>
```

- „Einfache“ Variante der Criteria-API
- Nicht typsicher

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Kunde> cq = cb.createQuery(Kunde.class);
Root<Kunde> kunde = cq.from(Kunde.class);
Path<String> name = kunde.get("name");
cq.select(kunde).
    where(cb.in(name).value("Müller")
        .value("Huber").value("Mayer"));
Query query = em.createQuery(cq);
```

```
@Entity
public class Kunde {
    @NotNull
    @Size(min=2,max=25)
    String name;
    @NotNull
    @Size(min=2,max=25)
    String vorname;
    @ElementCollection
    Set<Telefon> telefonnummern;
}
```

```
@StaticMetamodel(Kunde.class)
public class Kunde_ {
    public static volatile
        SingularAttribute<Kunde, String> name;
    public static volatile
        SingularAttribute<Kunde, String> vorname;
    public static volatile
        SetAttribute<Kunde, Telefon> telefonnummern;
}
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Kunde> cq = cb.createQuery(Kunde.class);
Root<Kunde> kunde = cq.from(Kunde.class);
Path<String> name = kunde.get(Kunde_.name);
cq.select(kunde).
    where(cb.in(name).value("Müller").value("Huber").value("Mayer"));
Query query = em.createQuery(cq);
```

//Äquivalentes Query via JPA-QL

```
Query q = em.createQuery(
    "select k from Kunde k where k.name in ('Müller','Huber','Mayer')");
```

# Entity Callbacks

- Java Persistence API stellt für die aktive Steuerung des Lebenszyklus von Persistence Entities folgende Meta-Informationen zur Verfügung:
  - `@javax.persistence.PrePersist`
    - tritt vor dem Aufruf von `EntityManager.persist( )` auf
  - `@javax.persistence.PostPersist`
    - tritt nach dem Aufruf von `EntityManager.persist( )` auf
  - `@javax.persistence.PreRemove`
    - tritt vor dem Aufruf von `EntityManager.remove( )` auf
  - `@javax.persistence.PostRemove`
    - tritt nach dem Aufruf von `EntityManager.remove( )` auf
  - `@javax.persistence.PreUpdate`
    - tritt vor der Synchronisation von Änderungen in die Datenbank auf
  - `@javax.persistence.PostUpdate`
    - tritt nach der Synchronisation von Änderungen in die Datenbank auf
  - `@javax.persistence.PostLoad`
    - tritt nach dem Aufruf von `manager.find( )` oder `manager.getReference( )` auf

```
package de.mathema.domain;
...
@Entity
public class Customer extends User implements Serializable{
    public enum Type{PRIVATE, ORGANISATION}

    private CreditCard creditCard;
    private Type type;
    private Collection<Phone> phones = new ArrayList<Phone>();
    private Collection<Reservation>
        reservations = new ArrayList<Reservation>();

    ...

    @PostPersist
    protected void afterPersist(){
        System.out.println("After Persist");
    }

    @PostLoad
    protected void afterLoad(){
        System.out.println("After Load");
    }
}
```

- ▼ EntityListener kann auf definierten Ereignissen des Lebenszyklus einer Entity angebracht werden
  - `@PrePersist` – wenn die Anwendung `persist()` aufruft
  - `@PostPersist` – nach dem SQL `INSERT`
  - `@PreRemove` – wenn die Anwendung `remove()` aufruft
  - `@PostRemove` – nach dem SQL `DELETE`
  - `@PreUpdate` – wenn Container feststellt, daß eine Instanz „dirty“ ist
  - `@PostUpdate` – nach dem SQL `UPDATE`
  - `@PostLoad` – nach dem Aufladen einer Instanz

## ▼ EntityListeners Annotation

```
package javax.persistence;  
public @interface EntityListeners {  
    Class[] value();  
}
```



```
package de.mathema.listeners;
...
public class AuditListener {
    @PostPersist
    void postInsert(Object object){
        System.out.println("Inserted entity: " + object.getClass().getName());
    }
    @PostLoad
    void postLoad(Object object){
        System.out.println("Loaded entity: " + object.getClass().getName());
    }
}
```

```
package de.mathema.domain;

@Entity
@EntityListeners({AuditListener.class})
public class Customer extends User implements Serializable{
    public enum Type{PRIVATE, ORGANISATION}
    private CreditCard creditCard;
    ...
}
```

```
<entity class="de.mathema.domain.Customer">
  <entity-listeners>
    <entity-listener
      class="de.mathema.listeners.AuditListener">
    </entity-listener>
  </entity-listeners>
</entity>
```

# Transaktionen

- ▼ Transaktion (TX) involviert üblicherweise einen Austausch zwischen zwei Parteien
- ▼ **ACID**: die vier Prinzipien des Transaktionsservices
  - **Atomarität**: **TX** wird entweder ganz oder gar nicht ausgeführt
  - **Konsistenz**: konsistenter Datenzustand nach Beendigung der **TX**
  - **Isolation**: **TX** beeinflussen sich nicht gegenseitig
  - **Durabilität**: das Ergebnis einer **TX** ist dauerhaft
- ▼ besteht aus einem oder mehreren DB Zugriffen
  - Rollback
    - wenn einer der Zugriffe scheitert
    - durch einen programmatischen Einsatz
  - Commit
    - wenn alle Zugriffe erfolgreich verlaufen

## ▼ Container-Managed Transaction

- spezifiziere Enterprise Bean mit der Annotation `@javax.ejb.TransactionManagement` (`javax.ejb.TransactionManagementType.CONTAINER`)
- Transaktionsabgrenzungen können durch die Annotation `@javax.ejb.TransactionAttribute` oder im XML Deployment Deskriptor gesetzt werden

## ▼ Bean-managed Transaction

- spezifiziere Enterprise Bean mit der Annotation `@javax.ejb.TransactionManagement` (`javax.ejb.TransactionManagementType.BEAN`)
- Transaktionsabgrenzungen werden programmatisch gelöst

## ▼ Client-managed Transaction

- Transaktion wird vom Client gestartet/committed
- Kontext der Transaktion breitet sich auf die Bean aus

- ▼ Transaktionen können vom Container am Anfang/Ende einer Methode gestartet/committed werden
- ▼ Transaktionsverhalten von EJBs kann durch das Setzen von Attributen in die Annotation `@javax.ejb.TransactionAttribute` oder im EJB Deployment Deskriptor kontrolliert werden
- ▼ Transaktionsattribute sind:
  - `NotSupported`
  - `Supports`
  - `Required`
  - `RequiresNew`
  - `Mandatory`
  - `Never`

```
package javax.ejb;

public enum TransactionAttributeType {
    MANDATORY,
    REQUIRED,
    REQUIRES_NEW,
    SUPPORTS,
    NOT_SUPPORTED,
    NEVER
}
```

```
package javax.ejb;

public @interface TransactionAttribute {
    TransactionAttributeType value()
        default TransactionAttributeType.REQUIRED;
}
```

## ▼ Zwei Typen von Exceptions:

- System Exceptions (unchecked Exceptions)
  - Exceptions und Fehler, die von Systemsdiensten verursacht werden
    - `java.lang.NullPointerException`
    - `java.lang.IndexOutOfBoundsException`
    - `java.lang.OutOfMemoryError`
    - `javax.transaction.RollbackException`
    - ...
- Application Exceptions (checked Exceptions)
  - Exceptions, die von der Geschäftslogik geworfen werden
    - `IncompleteConversationalState`
    - `PaymentException`



- ▼ Alle **RuntimeExceptions**, **Errors** und unerwartete Exceptions aus dem Applikation Server oder Ressourcen
- ▼ Bean Provider Verantwortlichkeiten
  - **Errors** und **RuntimeExceptions** müssen im Container verbreitet werden
  - Andere Exceptions müssen in einer neuen **EJBException** abgefangen und verpackt werden
- ▼ Applikation Server Verantwortlichkeiten
  - Kommunikation zu den Clients als **RemoteException**
  - automatisches Rollback der Transaktion
  - Bean Instanz wird abgeworfen
- ▼ **RuntimeException** und **RemoteException** Unterklassen können durch die Nutzung der Annotation **@javax.ejb.ApplicationException** in Applikation Exceptions umgeformt werden

- ▼ Methoden der Geschäftslogik
  - benutzerdefinierte Exceptions
- ▼ werden unverändert an den Client weitergegeben
  - müssen im Remote Business Interface deklariert werden
- ▼ kein automatisches Rollback der Transaktion
  - muss manuell durchgeführt werden, falls die Zustände einer Bean inkonsistent bleiben
- ▼ **@javax.ejb.ApplicationException**
  - kann verwendet werden, um eine Application Exception zu erzwingen und die Transaktion automatisch zum Rollback zu führen

```
package javax.ejb;
public @interface ApplicationException{
    boolean rollback() default false;
}
```

```
@ApplicationException(rollback=true)
public class ProcessPaymentException extends Exception {
    public ProcessPaymentException() {
        super();
    }
    public ProcessPaymentException(String message) {
        super(message);
    }
}
```

```
<ejb-jar>
  <assembly-descriptor>
    <application-exception>
      <exception-class>
        java.sql.SQLException</exception-class>
      <rollback>true</rollback>
    </application-exception>
  </assembly-descriptor>
</ejb-jar>
```

- ▼ **java.sql.SQLException** kann jetzt als Application Exception benutzt werden – sie verursacht ein Rollback