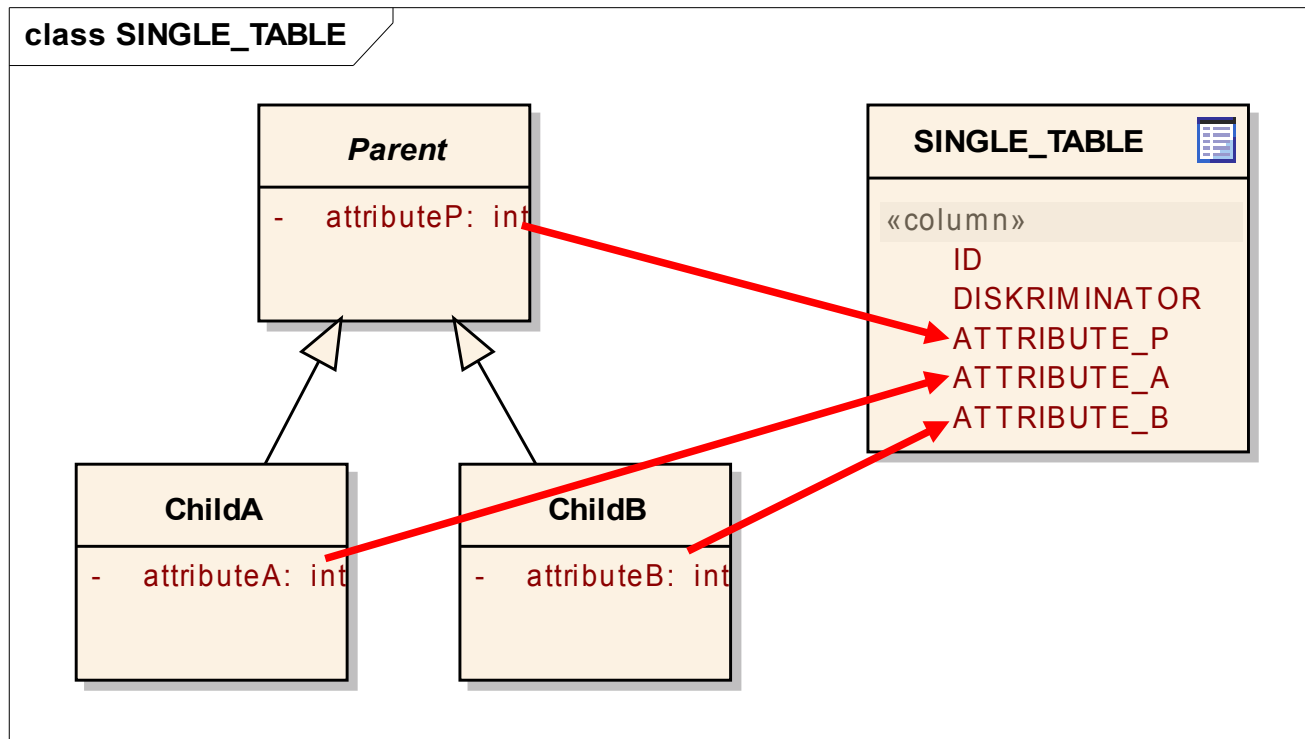


JPA / Hibernate



Alexander Kunkel

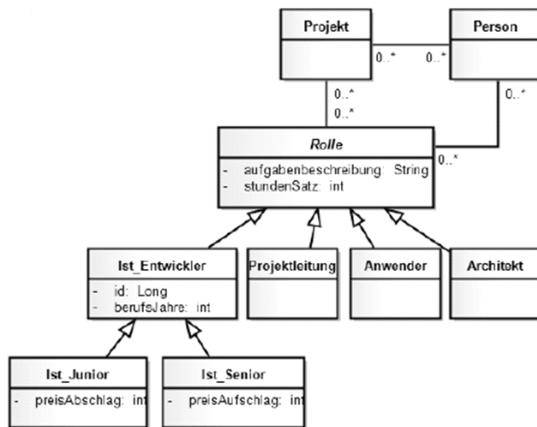
<http://www.kunkelgmbh.de/jpa/jpa.html>

jpa@kunkelgmbh.de

JPA Praxiswissen mit Hibernate

Alexander Kunkel

JPA Praxiswissen mit Hibernate



SHAKER
media

- Die ideale Ergänzung zu den Folien
- Detailliert erläuterte Beispiele
- Glossar mit Begriffsdefinitionen
- Stichwortverzeichnis

- ISBN: 978-3-86858-276-5
- 252 Seiten, 22,90€
- Beim Verlag Shaker Media:
<http://www.shaker-media.eu/de/content/bookshop/index.asp?ISBN=978-3-86858-276-5&ID=2>
- Bei Amazon:
<http://www.amazon.de/JPA-Praxiswissen-Hibernate-Alexander-Kunkel/dp/3868582762/>
- Aktuelles Inhaltsverzeichnis:
<http://www.kunkelgmbh.de/jpa/jpa.html>

Inhaltliche Schwerpunkte

- Wissen wird anhand aufeinander aufbauender praktischer Problemstellungen mit Beispielen aufgebaut.
- Primär JPA, nicht Hibernate proprietär. In Büchern zu Hibernate steht in der Regel die mächtigere aber proprietäre API von Hibernate im Mittelpunkt.
- Einsatz in JSE nicht JEE.
- Funktionale Aspekte stehen im Vordergrund.
- Schrittweise Einführung unter Vermeidung zu vieler Details.
- Die Beispiele zeigen auch die resultierenden DB-Statements.
- Kein XML-Mapping, ausschließlich Annotationen
- Das Thema Konfiguration wird nicht angesprochen.

Literatur

- „Java Persistence With Hibernate“ (deutsche Fassung)
 - Hanser
 - Christian Bauer, Gavin King
- „Java-Persistence-API mit Hibernate“
 - Addison Wesley
 - Bernd Müller, Harald Wehr
- EJB Spezifikation
 - <http://java.sun.com/products/ejb/docs.html>
- Hibernate Dokumentation
 - www.hibernate.org
- „Hibernate und die Java Persistence API“
 - Entwickler Press
 - Robert Hien, Markus Kehle
- Java Persistence And EJB3 (Vortrag)
 - Linda DeMichiel, chief architect EJB3
 - <http://www.infoq.com/presentations/ejb-3>

Neuerungen der JPA 2.0. Das aus meiner Sicht Nützlichste zuerst.

- Integration der JSR-303 Bean Validation
- Verschiedene Detailverbesserungen:
 - Pessimistisches Locken
 - Detach von einzelnen Objekten
 - Abfragen, die nicht polymorph sind, sondern sich auf konkrete Entity-Klassen beziehen.
 - @Access
- Neue Mappingmöglichkeiten:
 - @OrderColumn
 - orphanRemoveal
 - Element-Collection
 - Mapping von Maps: Viele Varianten mit kleinen aber feinen Unterschieden.
 - Embeddeds können Beziehungen zu anderen Embeddeds haben. Leider können weiterhin Entities und Embeddeds ihre Rolle nicht je Beziehung tauschen.
- Criteria-API
- Metamodell
- Cache

Inhalt

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

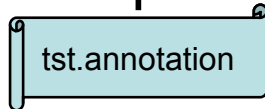
Optimierungsmöglichkeiten

Sperrstrategien

Designempfehlungen

Legende

- Beispielsourcecode im Package `tst.annotations`



- Hibernate Feature, nicht in JPA



- Nur JPA 1.0



- Ab JPA 2.0



- Ab JPA 2.1 (draft)



Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

Arbeitsumgebung

Datenbank HSQL

HSQLDB

Starten der HSQL Datenbank.

Quantum DB - Adresse.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

1 Headquarter
2 AgentServer
3 HSQL DatabaseManager
4 Generate DDL
5 Start HSQL Server

Debug As
Debug...
Organize Favorites...

Quantum SQL Queries View

Quantum Table View | Quantum SQL Log | Console x

No consoles to display at this time.

Quantum DB - Adresse.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

Database Bookmarks x

Agent-DB lokal (point)
Agent-DB lokal (mysql)
HSQL-Test
Quick List
Recent SQL Statements
INFORMATION_SCHEMA
PUBLIC
Views
Tables
B_PERSON
C_PERSON
EM_PERSON
FK2_ADRESSE
FK2_PERSON
FK_ADRESSE
FK_PERSON
HIBERNATE_SEQUENCES
L_PERSON
LE_PERSON
LE_TELEFON
O2M_PERSON
O2M_PERSON_O2M_TELEFON
O2M_TELEFON

Quantum SQL Queries View

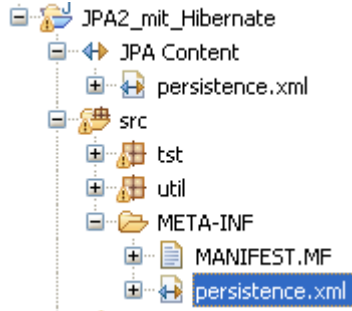
Quantum Table View | Quantum SQL Log | Console x

```
Start HSQL Server [Java Application] E:\Zentrale Entwicklungsumgebung\jdk1.5.0_09\bin\javaw.exe (15.01.2007 05:
[Server@13c6641]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@13c6641]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@13c6641]: Startup sequence initiated from main() method
[Server@13c6641]: Loaded properties from [E:\Zentrale Entwicklungsumge
[Server@13c6641]: Initiating startup sequence...
[Server@13c6641]: Server socket opened successfully in 0 ms.
[Server@13c6641]: Database [index=0, id=0, db=file:test, alias=] opene
[Server@13c6641]: Startup sequence completed in 2860 ms.
[Server@13c6641]: 2007-01-15 05:16:11.531 HSQLDB server 1.8.0 is onlin
[Server@13c6641]: To close normally, connect and execute SHUTDOWN SQL
[Server@13c6641]: From command line, use [Ctrl]+[C] to abort abruptly
```

Connected to HSQL-Test



Die für die Arbeitsumgebung gewählte Konfigurationsmöglichkeit (JPA 2)



Liste der Entry-Klassen wird von Eclipse automatisch gepflegt

Datenbank-Verbindung: Hier schon mit den JPA 2 Property-Namen.

Datenbank-dialekt

Statements auf der Console anzeigen

DB-Schema automatisch erzeugen

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="test">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>tst.validation.V_Person</class>
    [...]
    <properties>
      <!-- Einstellungen für die JDBC-Connection -->
      <property name="javax.persistence.jdbc.driver"
        value="org.hsqldb.jdbcDriver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:hsqldb:hsqldb://localhost" />
      <property name="javax.persistence.jdbc.user" value="sa" />
      <property name="javax.persistence.jdbc.password" value="" />

      <!-- SQL Dialekt -->
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect" />

      <!-- SQL-Anweisungen auf der Console ausgeben -->
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
  </persistence-unit>
</persistence>
```

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

Designempfehlungen

- Metadaten direkt im Sourcecode hinterlegen.
- Diese Metadaten können zur Compilezeit oder zur Laufzeit ausgelesen werden.
- Annotationen können sich auf folgende Elemente beziehen:
 - Packages
 - Klassen
 - Interfaces
 - Enumerations
 - Methoden
 - Variablen
 - Methodenparameter
- Es gibt vordefinierte Annotationen
 - `@Deprecated`
 - `@Override`
 - `@SuppressWarnings`

Java Annotationen

Eigene Annotation

@Length

- Annotation @Length
 - Längenangabe von Stringfeldern als Annotation.
- MyClass
 - Testklasse mit annotierten Stringfeldern.

Annotation @Length

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Length {
    int max();
}
```

SOURCE
CLASS
RUNTIME

TYPE
FIELD
METHOD
PARAMETER
CONSTRUCTOR
LOCAL_VARIABLE
ANNOTATION_TYPE
PACKAGE

MyClass

```
public class MyClass {
    String s1;

    @Length(max = 32)
    String s2;

    @Length(max = 15)
    String s3;

    public String toString() {
        return "s1='" + s1 ...
    }
}
```

tst.annotation.length

- LengthChecker
 - Eine Prüffunktion, die annotierte Stringfelder eines Objekts auf ihre Länge hin überprüft.

LengthChecker

```
public class LengthChecker {  
  
    public void check(Object obj) {  
        ArrayList<Field> annotatedFields;  
        try {  
            annotatedFields = getLengthAnnotatedFields(obj);  
            for (Field field : annotatedFields) {  
                Length length = field.getAnnotation(Length.class);  
                String s = (String) field.get(obj);  
                if (s != null && s.length() > length.max()) {  
                    System.out.println("Field '"  
                        + field.getName()  
                        + "' is too long.");  
                }  
            }  
        }  
        catch (Exception e) {  
        }  
    }  
    [...]  
  
    private ArrayList<Field> getLengthAnnotatedFields(Object obj)  
    [...]
```

tst.annotation

- TestLength
 - Das passende Testprogramm dazu.

TestLength

```
public class TestLength {  
  
    public static void main(String[] args) {  
        MyClass mc = new MyClass();  
  
        mc.s1 = "Donald Duck is a JPA evangelist.";  
        mc.s2 = "Donald Duck is a JPA evangelist.";  
  
        LengthChecker checker = new LengthChecker();  
        checker.check(mc);  
        checker.preserve(mc);  
  
        System.out.println(mc);  
    }  
}
```

Field 's2' is too long.

Schneidet s2 auf die
annotierte Länge ab.

Konsole

Field 's2' is too long.

s1='Donald Duck is a JPA evangelist.',
s2='Donald Duck is '

tst.annotation.length

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

Designempfehlungen

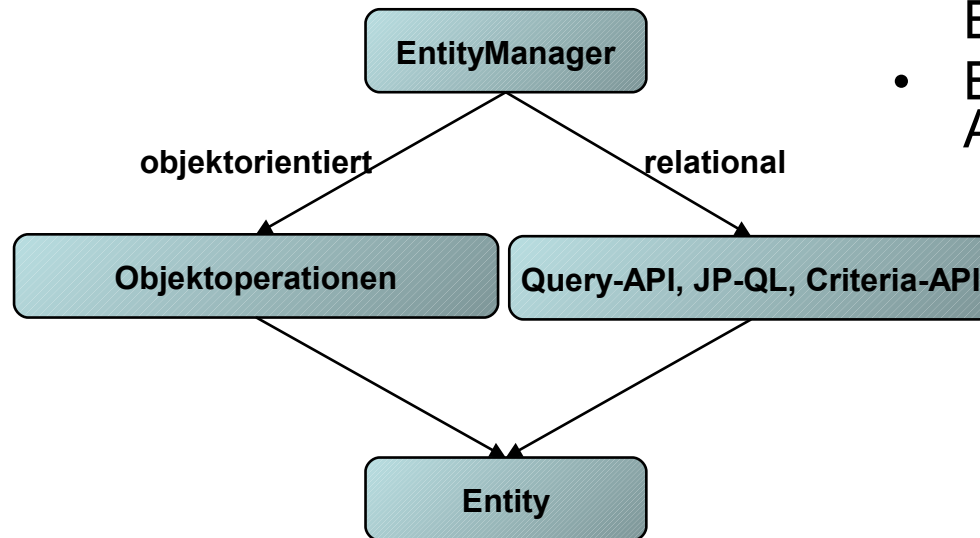
Java Persistence API - Einführung

Beteiligte Komponenten

EntityManager

Entity

EJB QL, Query-API



- Zugang zu den Datenbankfunktionalitäten erhält man mittels dem EntityManager.
- Es gibt 2 unterschiedliche Arbeitsweisen:
 - Eher objektorientiert mit den Operationen „persist“, „update“, „remove“, „find“ und „merge“.
 - Eher relational mit der Datenbanksprache Query-API und der JP-QL oder seit JPA 2 mit einer programmierten Abfrage anhand der Criteria API **2.0**.
- Beide Arbeitsweisen beziehen sich letztendlich auf Datenobjekte → Entities und deren Life-Cycle

Java Persistence API - Einführung

Beteiligte Komponenten

Vergleich
Objektoperationen
JP-QL, Query-API

Objektoperationen

- Entityobjekte stehen im Mittelpunkt der Datenbankoperationen: persist, remove, merge, find
- Fortsetzung der Operationen über Beziehungen hinweg (Transitive Persistenz) → Cascade
- Ownership

JP-QL, Query-API, Criteria-API

- Datenbanksprache JP-QL steht im Mittelpunkt der Query-API: select, update, delete
- Alternativ zur textuellen Formulierung einer Abfrage mit JP-QL ist es seit JPA 2 mittels Criteria-API möglich, diese auch programmatisch zu erstellen.
- Entityobjekte in Abfrageergebnissen.

Java Persistence API - Einführung

Entity

Entity

Markieren der Klasse als „Entity“. Ohne weitere Angaben ist der Name der Tabelle der selbe wie der der Klasse.

Attribut „Name“ als Primärschlüssel.

Attribute werden ohne weitere Angaben automatisch in gleichnamige Tabellenspalten gespeichert.

PERSON
NAME : VARCHAR(255)
VORNAME : VARCHAR(255)

```
@Entity
public class Person {
    @Id
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    private String vorname;

    public String getVorname() {
        return vorname;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
}
```

- Benötigt einen Defaultkonstruktor.
- Eine Entity-Klasse muss eine Top-Level-Klasse sein.
- Weder die Klasse noch seine Attribute und Methoden dürfen ‚final‘ sein.
- Die für die Persistenz relevanten Attribute müssen public Getter- und Settermethoden gemäß der Java-Beans Spezifikation haben.
- Eine Entity muss einen Primärschlüssel haben.
- Die Annotationen können alternativ an den Objektvariablen oder den Getter-/Settermethoden notiert werden. Field-based access, Property-based access.
- Mischen von Field-based und Property-based ist seit JPA 2 mit der Annotation **@Access** ebenso möglich.

2.0

tst.firstentity

EntityManager: Methoden, die in den Beispielen verwendet werden

```
// Return the resource-level transaction object. The EntityTransaction
// instance may be used serially to begin and commit multiple transactions.
public EntityTransaction getTransaction();

// Make an instance managed and persistent.
public void persist(Object entity);

// Remove the entity instance.
public void remove(Object entity);

// Find by primary key.
public <T> T find(Class<T> entityClass, Object primaryKey);

// Get an instance, whose state may be lazily fetched.
public <T> T getReference(Class<T> entityClass, Object primaryKey);

// Create an instance of Query for executing a Java Persistence query
// language statement.
public Query createQuery(String qlString);

// Close an application-managed EntityManager. After the close method has
// been invoked, all methods on the EntityManager instance and any Query
// objects obtained from it will throw the IllegalStateException except for
// getTransaction and isOpen (which will return false). If this method is
// called when the EntityManager is associated with an active transaction,
// the persistence context remains managed until the transaction completes.
public void close();
```

- .getTransaction()
 - .begin()
 - .commit()
 - .rollback()
- Leseoperationen können außerhalb einer Transaktion erfolgen.
 - .find()
 - .getReference()
 - .refresh()
- Einige modifizierende Operationen können ebenfalls außerhalb einer Transaktion gerufen werden. Veränderungen werden in einer Warteschlange zurückgehalten, bis eine Transaktion eröffnet wird. → !!! Keine Exception
 - .persist()
 - .merge()
 - .remove()
- Einige Operationen können ausschließlich innerhalb einer Transaktion gerufen werden. → TransactionRequiredException
 - .flush()
 - .lock()
 - Abfragen mit update/delete.

- Leichtgewichtige Session zur Datenbank mit 2 möglichen Lebensdauern
 - STANDARD: Für die Dauer einer einzigen Transaktion gültig
 - EXTENDED: Für die Dauer mehrerer Transaktionen gültig
- **!!! Nicht threadsafe !!!**
→ Mehrere parallele Transaktionen nur mittels mehreren parallelen EntityManagern möglich.
- Fehler werden mittels Runtime-Exceptions transportiert.
- Sobald eine Exception während dem commit auftritt, muss wenigstens ein rollback auf der Transaktion erfolgen, sonst kann man mit dem EntityManager nicht mehr sinnvoll weiterarbeiten.

Rahmen für eine Transaktion in einer JSE Umgebung:

```
EntityManager em = ...
EntityTransaction tx = null;
try {
    tx = em.getTransaction();
    tx.begin();
    // do some work
    [...]
    tx.commit();
}
catch (PersistenceException e) {
    if ( tx != null && tx.isActive() )
        tx.rollback();
    throw e; // or display error message
}
finally {
    em.close(); // optional
}
```

Person

```
@Entity
public class Person {
    @Id
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    private String vorname;

    public String getVorname() {
        return vorname;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
}
```

Insert

```
EntityManager em = JpaUtil.getEntityManager();

Person person = new Person();
person.setName("Duck");
person.setVorname("Donald");

em.getTransaction().begin();
em.getTransaction().persist(person);
em.getTransaction().commit();
```

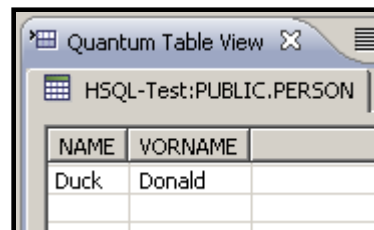
Entity-Objekt erzeugen

Objekt speichern

Hibernate-Log

```
insert into Person (vorname, name) values (?, ?)
```

DB



NAME	VORNAME
Duck	Donald

Person

```
@Entity
public class Person {
    @Id
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    private String vorname;

    public String getVorname() {
        return vorname;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
}
```

Update

```
EntityManager em = JpaUtil.getEntityManager();

Person person = em.find(Person.class, "Duck");
person.setVorname("Dagobert");

em.getTransaction().begin();
em.getTransaction().commit();
```

Objekt anhand
Primärschlüssel laden

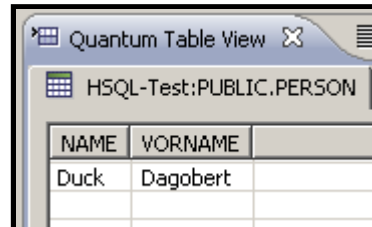
Setze neuen Vornamen

Entity-Objekt speichern,
em.persist(...) ist unnötig

Hibernate-Log

```
select person0_name ... where person0_name=?
update Person set vorname=? where name=?
```

DB



NAME	VORNAME
Duck	Dagobert

Java Persistence API - Einführung

Entity

DB-Delete

Person

```
@Entity
public class Person {
    @Id
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    private String vorname;

    public String getVorname() {
        return vorname;
    }

    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
}
```

Delete

```
EntityManager em = JpaUtil.getEntityManager();

Person person = em.getReference(Person.class, "Duck");

em.getTransaction().begin();
em.remove(person);
em.getTransaction().commit();
```

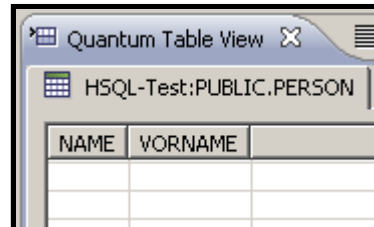
Objektreferenz anhand
Primärschlüssel laden

Objekt in DB löschen

Hibernate-Log

```
select person0_name ... where person0_name=?
delete from Person where name=?
```

DB



NAME	VORNAME

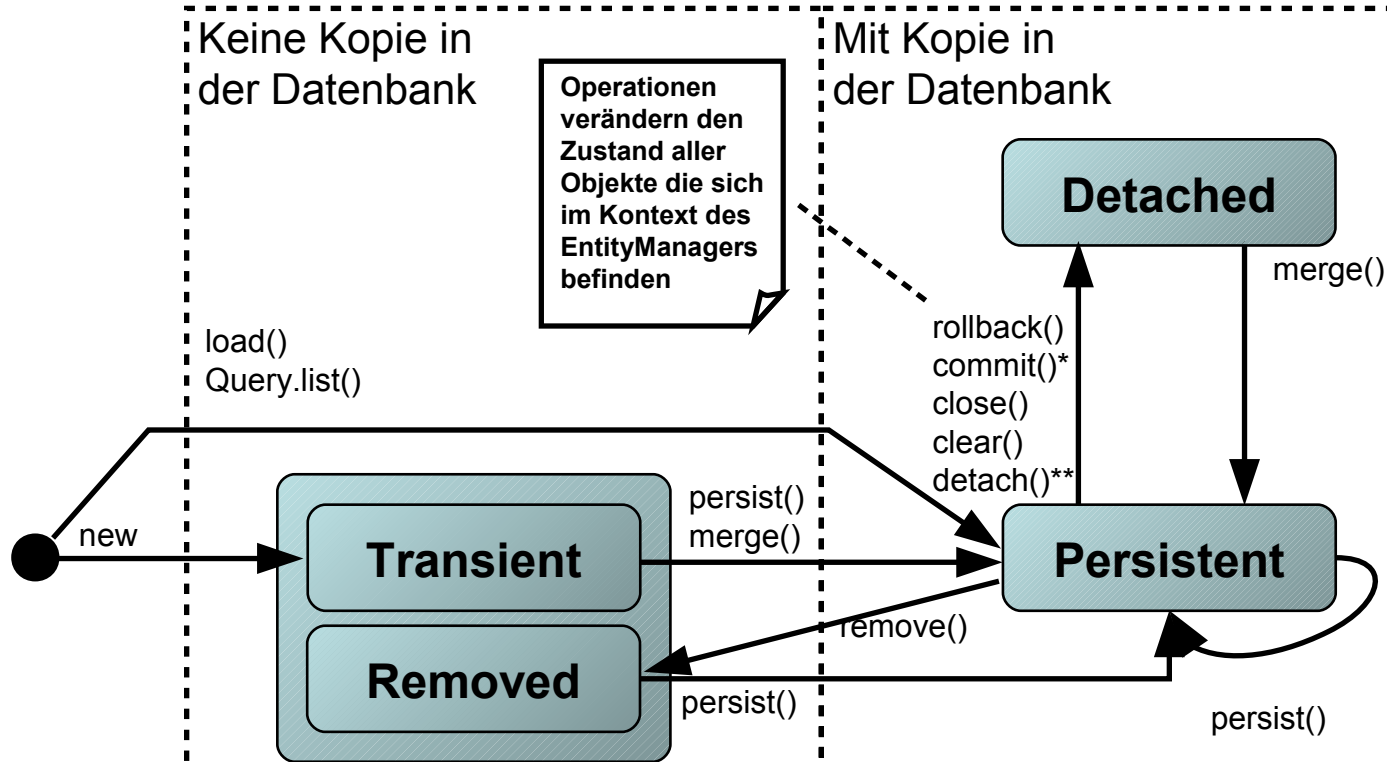
Select erfolgt, obwohl
das Object lediglich
mittels ‚getReference‘
adressiert wird.

tst.firstentity

Java Persistence API - Einführung

Entity - Life-Cycle

Transient
Persistent
Detached



* Gilt nur für den STANDARD EntityManager

** Ab JPA 2

Java Persistence API - Einführung

Entity - Life-Cycle

Transient
Persistent
Detached

Transient (vorübergehend, flüchtig)

Mit ‚new‘ erzeugte Objekte befinden sich zunächst lediglich im Arbeitsspeicher. Dieser Zustand wird als ‚Transient‘ bezeichnet. Durch Übergabe an ‚persist‘ des Entity-Managers wechselt der Zustand zu ‚Persistent‘.

Zu dem Objekt gibt es keinen korrespondierenden Datensatz in der Datenbank.

Persistent (nicht flüchtig)

Mit ‚new‘ erzeugte Objekte befinden sich zunächst lediglich im Arbeitsspeicher. Dieser Zustand wird als ‚Transient‘ bezeichnet. Durch Übergabe an ‚persist‘ des Entity-Managers wechselt der Zustand zu ‚Persistent‘.

Detached (gelöst)

Vom Persistenzkontext ‚gelöst‘. Im Unterschied zu ‚Transient‘ war das Objekt persistent und hat noch den Primärschlüssel, zu dem es immer noch einen Datensatz in der Datenbank gibt.

Removed (gelöscht)

Vom Persistenzkontext ‚gelöst‘. Im Unterschied zu ‚Transient‘ war das Objekt persistent und hat noch den Primärschlüssel. Ein Datensatz in der Datenbank existiert allerdings nicht mehr.

Java Persistence API - Einführung

Entity - Life-Cycle

Detached
merge()

- Instanzen „außerhalb“ der Verwaltung eines EntityManagers.
- Bei em.merge immer nur mit den zurückgelieferten Objekten weiterarbeiten!
- Em.merge löst entsprechenden Select gegen die DB aus.
- Standardproblem bei Web-Anwendungen. Lösungsmuster siehe Hibernatedokumentation ...

Java Persistence API - Einführung

Entity – Auf den Life-Cycle reagieren

Was und wie?

- Mit Mitteln der JPA kann programmatisch auf den Zustandswechsel eines Datenobjektes reagiert werden.
 - Komplexe Validierung vor dem Speichern eines Datenobjektes
 - Automatische Pflege eines Einfüge- und Aktualisierungszeitstempels
- Verschiedene Möglichkeiten, auf den Life-Cycle zu reagieren:
 - Durch die Definition der Callback-Methoden in einer Entitätsklasse (Beispiel folgt).
 - Durch die Definition der Callback-Methoden bei Vererbung in der „mapped superclass“. Einzelne Ausnahmen können bei Bedarf mit der Annotation `@ExcludeSuperclassListeners` gemacht werden.
 - Durch die Definition eines Entity-Listeners mit Callback-Methoden (Beispiel folgt).
 - Durch die Definition eines Entity-Listeners als Standard-Listener für alle Entitätsklassen einer Persistence-Unit im Deployment-Deskriptor `orm.xml`. Einzelne Ausnahmen können bei Bedarf mit der Annotation `@ExcludeDefaultListeners` gemacht werden.

Java Persistence API - Einführung

Entity – Auf den Life-Cycle reagieren

Callback-Methode

Person

```
@Entity
public class LC_Person {
    private Calendar inserted;

    public Calendar getInserted() {
        return inserted;
    }

    public void setInserted(Calendar inserted) {
        this.inserted = inserted;
    }

    @PrePersist
    protected void prePersist() {
        System.out.println("### prePersist()");
        inserted = Calendar.getInstance();
    }

    @PostPersist
    protected void nachDemSpeichern() {
        System.out.println("### nachDemSpeichern()");
    }

    @PreUpdate
    protected void preUpdate() {
        [...]
    }

    @PostUpdate
    protected void nachDemAktualisieren() {
        [...]
    }
}
```

Markiert die Callback-Methode, die vor einem INSERT gerufen werden soll.

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

LC_Person person = new LC_Person();
person.setName("Duck");
person.setVorname("Donald");

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
```

Hibernate-Log

```
### prePersist()
insert into LC_Person (inserted, updated, vorname, name) ...
### nachDemSpeichern()
```

Der Name einer Callback-Methode kann frei gewählt werden.

tst.lifecycle

Java Persistence API - Einführung

Entity – Auf den Life-Cycle reagieren

Listener

Person

```
@Entity
@EntityListeners(MyListener.class)
public class LC_Person {
    private Calendar inserted;

    public Calendar getInserted() {
        return inserted;
    }

    public void setInserted(Calendar inserted) {
        this.inserted = inserted;
    }

    [...]
}
```

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

LC_Person person = new LC_Person();
person.setName("Duck");
person.setVorname("Donald");

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
```

Hibernate-Log

```
### prePersist()
insert into LC_Person (inserted, updated, vorname, name) ...
### postPersist()
```

MyListener

```
public class MyListener {
    @PrePersist
    public void prePersist(LC_Person person) {
        System.out.println("### prePersist()");
    }

    @PostPersist
    public void postPersist(LC_Person person) {
        System.out.println("### postPersist()");
    }

    [...]
}
```

tst.lifecycle

Java Persistence API - Einführung

Entity – Auf den Life-Cycle reagieren

Aufrufreihenfolge

- Alle im XML-Deployment-Deskriptor angegebenen Standard-Listener werden in der Reihenfolge gerufen, wie sie dort definiert sind.
- Entity-Listener-Klassen, wie sie mittels der Annotation **@EntityListeners** angegeben wurden.
- Bei mehreren Entity-Listnern innerhalb einer Vererbungshierarchie haben die aus den Superklassen Vorrang.
- Die Callback-Methode, die in der Entitätsklasse selbst existiert.
- Der Ausschluss eines Listeners mittels **@ExcludeDefaultListeners** oder **@ExcludeSuperclassListeners** geht zum nächsten Listener über.
- Eine Exception innerhalb eine Listeners oder Callback-Methode bricht die Aufruf-Reihenfolge und die aktuelle Transaktion ab.

- Welche speziellen Vorgaben gibt es zu Callback-Methoden?
 - Die Sichtbarkeit (private, protected, public, package) kann beliebig eingestellt sein.
 - Die betreffende Methode darf nicht final oder static modifiziert sein.
 - Die Methodensignatur kann
 - innerhalb einer Entitätsklasse void <METHOD>() sein.
 - innerhalb eines Entity-Listeners void <METHOD>(Object) sein, wobei Object das betreffende Datenobjekt ist, das beim Aufruf von der Laufzeitumgebung übergeben wird.
- Was dürfen Sie innerhalb einer Callback-Methode tun und was nicht?
 - Es darf JDBC, JMS und EJB gerufen werden.
 - Es sollten nicht der EntityManager oder Funktionalitäten der Query gerufen werden.
 - Ebenso sollten die bestehenden Beziehungen nicht geändert werden. Diese Vorgabe könnte sich in einer späteren Version der Spezifikation noch ändern.

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

Designempfehlungen

Java Persistence API

Entity - Primärschlüssel

@Entity

@Id

@GeneratedValue

- Eine Entity muss einen Primärschlüssel haben.
- Markieren eines Feldes oder Methode mittels @Id als Primärschlüssel.
- Nicht zuletzt weil fachliche Schlüssel selten wirklich eindeutig sind, setzt man in der Regel auf technische Schlüssel.
- Technische Schlüssel (Surrogatschlüssel) lassen sich einfach mittels @GeneratedValue generieren.
- Es gibt mehrere Generatorstrategien.
- Zusammengesetzter Primärschlüssel ist mittels @IdClass möglich, aber ein wenig umständlich und lediglich für Legacy-Datenbanken relevant.

```
@Entity
public class PK_Telefon {
    @Id
    private String number;

    public String getNumber() {
        return number;
    }
}
```

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

    public Long getId() {
        return id;
    }
}
```

@GeneratedValue kann nur im Zusammenhang mit @Id benutzt werden.
Das schließt die Verwendung von @GeneratedValue für sonstige Ids leider aus.

tst.primarykey

Java Persistence API

Entity – Generator für Primärschlüssel

@GeneratedValue

- @GeneratedValue(strategy = ...) legt die Strategie zur Generierung des Primärschlüssels fest.
- JPA-Generatorstrategien:
 - GenerationType.AUTO – Wählt eine Strategie entsprechend der zugrunde liegenden Datenbank.
 - GenerationType.TABLE – Primärschlüssel werden mittels einer eigenen Tabelle verwaltet.
 - GenerationType.SEQUENCE – Nutzt eine Sequence, wie es sie beispielsweise in ORACLE gibt.
 - GenerationType.IDENTITY – Nutzt spezielle „Identity“ Spalten wie in MySql, HSQLDB.
- Es gibt noch weitere Hibernate Generatorstrategien

Java Persistence API

Entity – Eigener Generator

@GenericGenerator

Einfacher Generator

```
public class SimpleGenerator implements IdentifierGenerator {
    public synchronized Serializable generate(SessionImplementor session, Object object)
        throws HibernateException
    {
        return Long.valueOf(id++);
    }

    private static long id;
}
```



Person

```
@Entity
public class G_Person {
    private Long id;

    @Id
    @GeneratedValue(generator = "mysequence")
    @GenericGenerator(name = "mysequence", strategy = "tst.idgenerator.SimpleGenerator")
    public Long getId() {
        return id;
    }
}
```

Testcode

```
em.getTransaction().begin();

G_Person person = new G_Person();
person.setName("Bean");
em.persist(person);

em.getTransaction().commit();
```

DB

ID	NAME	VORNAME
0	Bean	<NULL>

tst.idgenerator

Java Persistence API

Entity - Grundlegende Annotationen

@Table
@Column
@Transient

- @Table
 - Definiert die primäre Tabelle für die Entity
 - Parameter: name, catalog, schema, uniqueConstraints
 - Fachlich motivierte Attributkombinationen können unabhängig vom technischen Schlüssel auf unique gesetzt werden.
- @Column
 - Definiert die Tabellenspalte für das markierte Feld
 - Parameter: name, unique, nullable, updateable, insertable, columnDefinition, table, length, precision, scale
- @Transient
 - Markiert ein nicht persistentes Feld

```
@Entity
@Table(name="T_ADRESSE")
public class Adresse {
    @Id
    private long id;

    @Column(name="C_STRASSE")
    private String strasse;

    @Transient
    private int temp;
}

create table T_ADRESSE (
    id bigint not null,
    C_STRASSE varchar(255),
    primary key (id)
);
```

```
@Entity
@Table(uniqueConstraints = {
    @UniqueConstraint(columnNames = {
        "name",
        "vorname"
    })
})
public class PK_Person {
    ...
}
```

tst.firstentity
tst.primarykey

Java Persistence API

Entity – Blobs und Clobs

@Lob

- Benutzen von BLOBS und CLOBS erfolgt unabhängig von der Datenbank.
- Für Java-Typen:
 - Byte[] (default)
 - byte[] (default)
 - Java.sql.Blob (default)
 - String
 - Character[]
 - char[]
 - java.sql.Clob (default)
- Bei Lobs wird aus dem Längenconstraint der konkrete Datenbanktyp abgeleitet. MySQL kennt für einen Blob unterschiedlich große Typen!

```
String lebenslauf;  
  
@Lob  
public String getLebenslauf() {  
    return lebenslauf;  
}  
  
public void setLebenslauf(String lebenslauf) {  
    this.lebenslauf = lebenslauf;  
}  
  
byte[] foto;  
  
@Column(length=65535)  
public byte[] getFoto() {  
    return foto;  
}  
  
public void setFoto(byte[] foto) {  
    this.foto = foto;  
}
```

Stellt man hier beispielsweise 65.536 ein, sorgt das bei MySQL dafür, dass Hibernate nicht den Defaulttyp: blob (65.536 bytes), sondern mediumblob (16.777.215 bytes) wählt.

tst.blobs

Java Persistence API

Vergleich – Blob vs. Byte[] (beim Einfügen)

Blob vs. Byte[]
(insert)

- Randbedingungen:
 - 250MB Textdatei
 - CPU ist inklusive DB
 - ORACLE 11g XE

	byte[]	Blob-Stream	JDBC-Array	JDBC-Stream
Laufzeit (s)	20	50	25	30
Max Memory (MB)	750	750	600	5
Durchschn. Memory (MB)	600	270	600	5
CPU-Auslastung (%)	80	60	65	60
CPU-Arbeitszeit (s)	16	30	16	18

tst.blobs

Java Persistence API

Entity - Datumsfelder

@Temporal

- Annotation kann an **java.util.Date** und **java.util.Calendar** angebracht werden
- Es gibt 3 Temporaltypen

```
public enum TemporalType {  
    DATE, //java.sql.Date  
    TIME, //java.sql.Time  
    TIMESTAMP //java.sql.Timestamp  
}
```

- Beispiel:

	JDBC-Typ	ORACLE-Typ	Beispielinhalt ORACLE
Date Temporal.DATE	java.sql.Date	date	2007-11-15 00:00:00.0
Date Temporal.TIME	java.sql.Time	date	2007-11-15 16:12:50.0
Date Temporal.TIMESTAMP	java.sql.Timestamp	timestamp	2007-11-15 16:12:50.203
Calendar Temporal.DATE	java.sql.Date	date	2007-11-15 00:00:00.0
Calendar Temporal.TIME	In Hibernate nicht implementiert	In Hibernate nicht implementiert	In Hibernate nicht implementiert
Calendar Temporal.TIMESTAMP	java.sql.Timestamp	timestamp	2007-11-15 16:12:50.203

Java Persistence API

Entity – Integer Typen

Integer Typen

Ein numerischer Integertyp wird durch Hibernate ohne Berücksichtigung von Precision bzw. Length mit fixer Länge auf die Datenbank gemappt. Mapping von Integertypen bei Oracle:

Java-Typ	JDBC-Typ	Oracle-Typ
java.lang.Byte	Types.TINYINT	number(3,0)
java.lang.Short	Types.SMALLINT	number(5,0)
java.lang.Integer	Types.INTEGER	number(10,0)
java.lang.Long	Types.BIGINT	number(19,0)

Java Persistence API

Entity – Enum Typen

@Enumerated

Person

```
public class E_Person {  
    [...]  
    @Id  
    private String name;  
  
    @Enumerated(EnumType.STRING)  
    private Familienstand familienstand1;  
  
    // default  
    @Enumerated(EnumType.ORDINAL)  
    private Familienstand familienstand2;  
    [...]  
}
```

DB

Quantum Table View Console
HSQL-Test:PUBLIC.E_PERSON

NAME	FAMILIENSTAND1	FAMILIENSTAND2	VORNAME
Duck	ledig	2	Donald

Testcode

```
EntityManager em = JpaUtil.getEntityManager();  
  
E_Person person = new E_Person();  
person.setVorname("Donald");  
person.setName("Duck");  
person.setFamilienstand1(Familienstand.ledig);  
person.setFamilienstand2(Familienstand.ledig);  
  
em.getTransaction().begin();  
em.persist(person);  
em.getTransaction.commit();
```

Enum

```
public enum Familienstand {  
    unbekannt, verheiratet, ledig, geschieden;  
}
```

tst.enumtype

Entity

DB-Constraints

Not Null
Length
Unique

Not Null

```
@Column(nullable = false)
public String getName() {
    return name;
}
```

Längenbeschränkung

```
@Column(length=30)
public String getStrasse() {
    return strasse;
}
```

Unique

```
@Column(unique = true)
public String getEmail() {
    return email;
}
```

```
@Entity
@Table(uniqueConstraints = { @UniqueConstraint(columnNames = { "name", "vorname" }) })
public class C_Person {
```

ORACLE DDL-Skript:

```
CREATE TABLE C_Person (
    id number(19,0) NOT NULL,
    name varchar2(255 CHAR) NOT NULL,
    vorname varchar2(255 CHAR),
    strasse varchar2(30 CHAR),
    email varchar2(255 CHAR) UNIQUE,
    PRIMARY KEY (id),
    UNIQUE (name, vorname)
);
```

Defaultbelegung von Datenbankfeldern

```
private String vorname = "Donald";

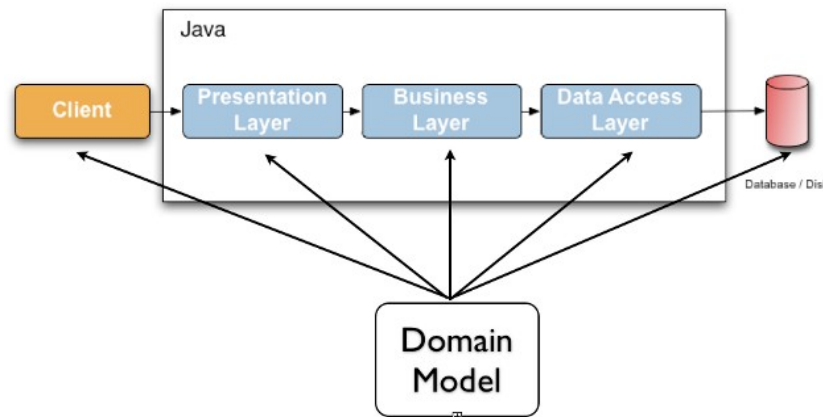
public String getVorname() {
    return vorname;
}
```

Foreign Key → Siehe „Beziehungen“

tst.constraints

- JSR-303 Bean Validation Framework

- Definiert Annotationen mittels derer POJOs um Validierungsinformationen ergänzt werden können.



Grafik entnommen aus: Hibernate Validator, JSR 303 Reference Implementation, Reference Guide, 4.1.0.Final

- Und JPA 2?

- Automatische Validierung der Datenobjekte (im Data Access Layer) vor dem Speichern in die Datenbank.
- Die automatische Validierung ist per Einstellung (in persistence.xml mittels `'javax.persistence.validation.mode'`) abschaltbar.
- Standardmäßig ist die automatische Validierung für INSERT und UPDATE aktiv, sofern eine JSR-303-Implementierung im Klassenpfad vorhanden ist.
- Ein erkannter Verstoß führt natürlich zum Abbruch der laufenden Transaktion.

JSR-303 Bean Validation

JSR-303 Validatoren

JSR-303 Validatoren Übersicht I

Annotation	Apply on	Use	Hibernate Metadata impact
@AssertFalse	field/property	Check that the annotated element is false.	none
@AssertTrue	field/property	Check that the annotated element is true.	none
@DecimalMax	field/property. Supported types are BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types.	The annotated element must be a number whose value must be lower or equal to the specified maximum. The parameter value is the string representation of the max value according to the BigDecimal string representation.	none
@DecimalMin	field/property. Supported types are BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types.	The annotated element must be a number whose value must be higher or equal to the specified minimum. The parameter value is the string representation of the min value according to the BigDecimal string representation.	none
@Digits(integer=, fraction=)	field/property. Supported types are BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types.	Check whether the property is a number having up to integer digits and fraction fractional digits.	Define column precision and scale.
@Future	field/property. Supported types are java.util.Date and java.util.Calendar.	Checks whether the annotated date is in the future.	none
@Max	field/property. Supported types are BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types.	Checks whether the annotated value is less than or equal to the specified maximum.	Add a check constraint on the column.
@Min	field/property. Supported types are BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types.	Checks whether the annotated value is higher than or equal to the specified minimum.	Add a check constraint on the column.

Entnommen aus: Hibernate Validator, JSR 303 Reference Implementation, Reference Guide, 4.1.0.Final

JSR-303 Bean Validation

JSR-303 Validatoren

JSR-303 Validatoren Übersicht II

Annotation	Apply on	Use	Hibernate Metadata impact
@NotNull	field/property	Check that the annotated value is not null.	Column(s) are not null.
@NotEmpty	field/property. Supported types are String, Collection, Map and arrays.	Check whether the annotated element is not null nor empty.	none
@Null	field/property	Check that the annotated value is null.	none
@Past	field/property. Supported types are java.util.Date and java.util.Calendar.	Checks whether the annotated date is in the past.	none
@Pattern(regex=, flag=)	field/property. Needs to be a string.	Checks if the annotated string matches the regular expression regex considering the given flag match.	none
@Size(min=, max=)	field/property. Supported types are String, Collection, Map and arrays.	Check if the annotated element size is between min and max (inclusive).	Column length will be set to max.
@ScriptAssert(lang=, script=, alias=)	type	Checks whether the given script can successfully be evaluated against the annotated element. In order to use this constraint, an implementation of the Java Scripting API as defined by JSR 223 ("Scripting for the Java™ Platform") must part of the class path. This is automatically the case when running on Java 6. For older Java versions, the JSR 223 RI can be added manually to the class path. The expressions to be evaluated can be written in any scripting or expression language, for which a JSR 223 compatible engine can be found in the class path.	none
@Valid	field/property. Any non-primitive types are supported.	Performs validation recursively on the associated object. If the object is a collection or an array, the elements are validated recursively. If the object is a map, the value elements are validated recursively.	none

Entnommen aus: Hibernate Validator, JSR 303 Reference Implementation, Reference Guide, 4.1.0.Final

JSR-303 Bean Validation

JSR-303 Validatoren

Zusätzl. Hibernate Validatoren



Annotation	Apply on	Use	Hibernate Metadata impact
@CreditCardNumber	field/property. The supported type is String.	Check that the annotated string passes the Luhn checksum test. Note, this validation aims to check for user mistake, not credit card validity!	none
@Email	field/property. Needs to be a string.	Check whether the specified string is a valid email address.	none
@Range(min=, max=)	field/property. Supported types are BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types.	Check whether the annotated value lies between (inclusive) the specified minimum and maximum.	none
@ScriptAssert(lang=, script=, alias=)	type	Checks whether the given script can successfully be evaluated against the annotated element. In order to use this constraint, an implementation of the Java Scripting API as defined by JSR 223 ("Scripting for the Java™ Platform") must part of the class path. This is automatically the case when running on Java 6. For older Java versions, the JSR 223 RI can be added manually to the class path. The expressions to be evaluated can be written in any scripting or expression language, for which a JSR 223 compatible engine can be found in the class path.	none
@URL(protocol=, host=, port=)	field/property. The supported type is String.	Check if the annotated string is a valid URL. If any of parameters protocol, host or port is specified the URL must match the specified values in the according part.	none

Entnommen aus: Hibernate Validator, JSR 303 Reference Implementation, Reference Guide, 4.1.0.Final

JSR-303 Bean Validation

Beispiel für eine manuelle Bean-Validierung

@Size
@Email
@Max, @Min

Beispielvalidierungen

```
@Size(min = 10, max = 30)
public String getName() {
    return name;
}
```

```
@Email
public String getEmail() {
    return email;
}
```



```
@Max(2006)
@Min(1900)
public Integer getGeburtsJahr() {
    return geburtsJahr;
}
```

Code zur Überprüfung auf Einhaltung der Validierungen

```
// Belegen der Attribute mit ungültigen Werten
V_Person person = new V_Person();
person.setVorname("Donald"); person.setName("Duck");
person.setEmail("abc"); person.setGeburtsJahr(1870);

// Erzeugen eines JSR-303 Validierers
ValidatorFactory
    factory = Validation.buildDefaultValidatorFactory();

Validator validator = factory.getValidator();

Set<ConstraintViolation<V_Person>>
    violations = validator.validate(person);

// Ausgabe der Reklamationen auf der Konsole
for (ConstraintViolation<V_Person> violation : violations) {
    System.out.println(violation.getMessage());
}
```

ORACLE DDL-Skript

```
create table V_Person (
    [...]
    geburtsJahr number(10,0)
        check (geburtsJahr>=1900
            and geburtsJahr<=2006),
    [...]
);
```

!!!Achtung!!!
Hier macht Hibernate in der Version 3.6.0Beta3 leider noch den Fehler, von @Max und @Min immer nur die @Max Annotation zu berücksichtigen.

Ergebnis aufgrund schwerer Verfehlungen :-)

muss zwischen 10 und 30 liegen
keine gültige E-Mail-Adresse
muss grössergleich 1900 sein

tst.validation

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

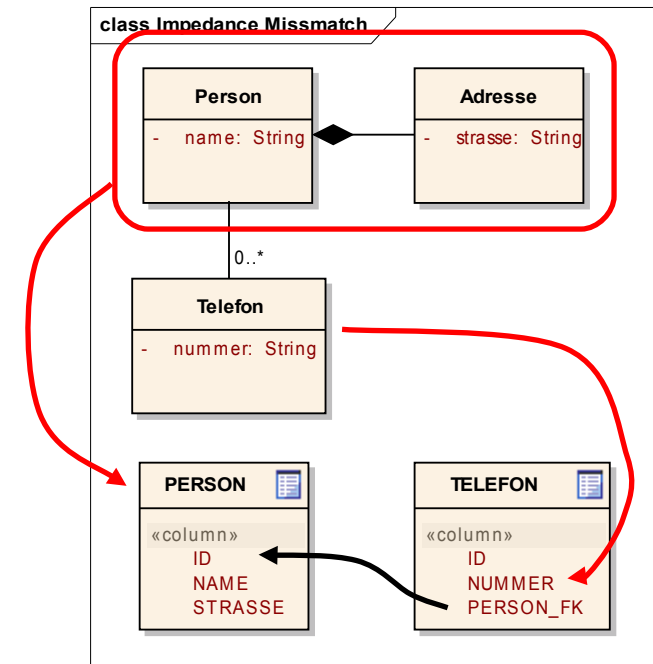
Designempfehlungen

Beziehungen

Paradigmen Mismatch

Granularität

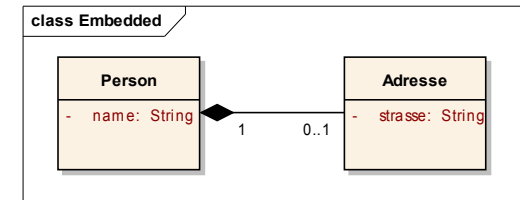
- Granularität von Objekten unterscheiden sich von Tabellen.
- Im Beispiel werden die Klassen Person und Adresse auf die Tabelle PERSON abgebildet.



1:1 Beziehung unidirektional

Varianten

- Anhand der 1:1 Beziehung lassen sich schon die meisten Konstrukte demonstrieren.
- 4 Möglichkeiten die 1:1 Beziehung zwischen Person und Adresse zu mappen
 - Embedded in eine Tabelle
 - Person und Adresse in getrennte Tabellen. Zusammengehörige Adresse und Person haben den selben Primärschlüssel.
 - Person und Adresse in getrennte Tabellen mit Foreign Key seitens Person auf Adresse Person und ggf. einen unique Constraint auf den Fremdschlüssel.
 - Person und Adresse in getrennte Tabellen mit Foreign Key seitens Adresse auf Person und ggf. einen unique Constraint auf den Fremdschlüssel.



Cascade (Transitive Persistenz)

CascadeType

- Objektoperationen werden mittels des EntityManagers ausgeführt.
- Die Fortsetzung der Operationen über Beziehungen hinweg kann eingestellt werden.
- Transitive Persistenz → Cascade
- Default: Ohne Angabe wird keine Operation über Beziehungen hinweg fortgeführt.
- CascadeType
 - PERSIST
 - MERGE
 - REFRESH
 - REMOVE
 - DETACH (JPA 2)
 - ALL
- Verwendbar bei:
@OneToOne, @OneToMany, @ManyToOne, @ManyToMany

tst.cascade



- Um was geht es?
 - Wer aufgrund referentieller Integrität das möchte, oder wer mittels JPA auf ein bestehendes Datenbankschema mit vorhandenen Cascade-Deletes zugreifen muss, der wird durch die JPA nicht unterstützt.
 - Hibernate füllt diese Lücke mit der eigenen Annotation `@OnDelete`.
 - `@OnDelete` kann an Beziehungen zu anderen Entitätsklassen und an einer Vererbungsbeziehung angebracht werden, sofern die beteiligten Klassen in unterschiedlichen Tabellen gespeichert werden.
- Wie wirkt die Annotation?
 - Beim löschen einer Person mit referenzierten Telefonnummern setzt Hibernate eine Delete-Anweisung für das betreffende Personenobjekt ab und verzichtet auf die Delete-Anweisung für die Telefonnummern.
 - Ein JPA-Cascade.REMOVE bzw. ALL an der Beziehung ist dennoch nötig, damit gleichzeitig auch die Telefonnummernobjekte aus dem Persistence-Context entfernt werden!
 - Beim Erzeugen eines DDL-Schemas wird der datenbankseitige Delete-Cascade erzeugt, sofern es die Datenbank überhaupt unterstützt.

DDL Delete Cascade Constraint

Beispiel Teil 1



Person

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "person")
@OnDelete(action=OnDeleteAction.CASCADE)
public List<CA_Telefon> getTelefonNummern() {
    if (telefonNummern == null) {
        telefonNummern = new ArrayList<CA_Telefon>();
    }

    return telefonNummern;
}
```

Kaskadiertes Löschen im Sinne der JPA ist dennoch notwendig!
Sonst sind die vermeintlich gelöschten Telefonnummern immer noch im Persistence-Context und führen dann zu diversen Problemen.

Die Annotation `@OnDelete` verhindert die Delete-Anweisung für die Telefonnummern, welche ja von der Datenbank automatisch gelöscht werden.

DDL-Script HSQLDB

```
[...]
alter table CA_Telefon
    add constraint FK2F2EB9DA3BD89228
    foreign key (person_id)
    references CA Person
    on delete cascade
[...]
```

tst.cascade



Testcode - Delete

```
EntityManager em = JpaUtil.getEntityManager();

CA_Person person = em.find(CA_Person.class, id);

// Telefonnummer und deren Primärschlüssel einer Telefonnummer für
// spätere Überprüfungen merken.
CA_Telefon telefon = person.getTelefonNummern().get(0);
Long telefonPk = telefon.getId();

em.getTransaction().begin();
System.out.println("### Ist Telefonnummer in der Obhut des EMs? "
    + em.contains(telefon));

em.remove(person);
System.out.println("### Ist Telefonnummer immer noch in der Obhut des EMs? "
    + em.contains(telefon));

System.out.println("### Ist Telefonnummer noch ladbar? "
    + em.find(CA_Telefon.class, telefonPk));

em.getTransaction().commit();
```

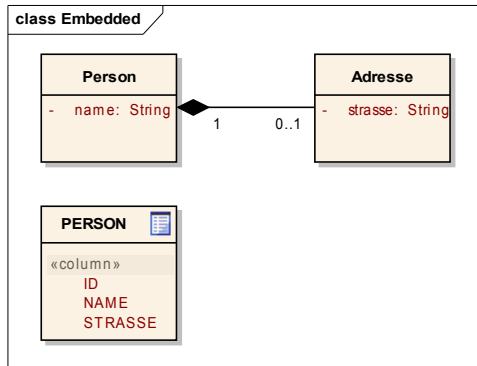
Hibernate-Log HSQLDB

```
select ca_person0_id as id40_0_, ... from CA_Person ca_person0_ where ca_person0_id=?
select telefonnum0_person_id as person3_40_1_, ... from CA_Telefon telefonnum0_ where telefonnum0_person_id=?
### Ist Telefonnummer in der Obhut des EMs? true
### Ist Telefonnummer immer noch in der Obhut des EMs? false
### Ist Telefonnummer noch ladbar? null
delete from CA_Person where id=?
```

tst.cascade

1:1 Beziehung unidirektional Embedded

@Embedded
@Embeddable



Person

```
private Em_Adresse adresse;

@Embedded
public Em_Adresse getAdresse() {
    return adresse;
}
```

Adresse

```
@Embeddable
public class Em_Adresse {
    ...
}
```

Der Life-Cycle von Adresse ist an den Life-Cycle von Person gekoppelt. Das heißt, dass beim Löschen der Person automatisch ihre Adresse gelöscht wird. Was bei Embeddeds leicht einzusehen ist, wenn Person und Adresse im selben Datensatz gespeichert sind. In der Objektorientierung spricht man bei existenziell gekoppelten Objekten von einer Komposition.

Bei einer Beziehung mit @OneToOne sind beide Life-Cycle getrennt.

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

Em_Person person = new Em_Person();
person.setVorname("Donald");
person.setName("Duck");

Em_Adresse adresse = new Em_Adresse();
adresse.setStrasse("Talstraße");
adresse.setHausnummer("15");

person.setAdresse(adresse);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
```

Hibernate-Log HSQLDB

```
insert into Em_Person (id, name, ...
call identity())
```

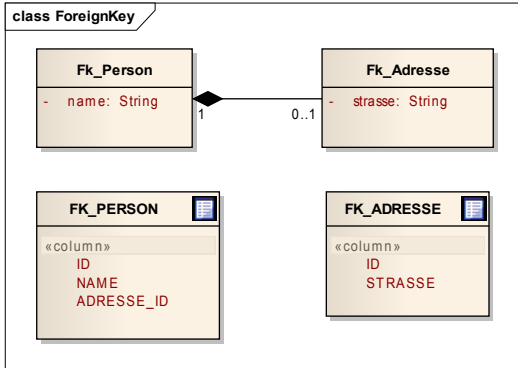
DB

ID	HAUSNUMMER	STRASSE	NAME	VORNAME
1	15	Talstraße	Duck	Donald

tst.one2one.embedded

1:1 Beziehung Standard

@OneToOne



Adresse

```
@Entity
public class Fk_Adresse {
    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }
    [...]
}
```

Person

```
@Entity
public class SPK_Person {
    private Long id;

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

    private SPK_Adresse adresse;

    @OneToOne(cascade=CascadeType.ALL)
    public SPK_Adresse getAdresse() {
        return adresse;
    }
    [...]
}
```

@OneToOne markiert die
1:1-Beziehung

tst.one2one.foreignkey

1:1 Beziehung

Standard (Fortsetzung)

@OneToOne

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

Fk_Person person = new Fk_Person();
person.setVorname("Donald");
person.setName("Duck");

Fk_Adresse adresse = new Fk_Adresse();
adresse.setStrasse("Talstraße");
adresse.setHausnummer("15");

person.setAdresse(adresse);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();

em.clear();

SFk_Person
    loadedPerson = em.find(Fk_Person.class, person.getId());

System.out.println(person.getAdresse().getStrasse());
```

ID	NAME	VORNAME	ADRESSE_ID
1	Duck	Donald	1

ID	HAUSNUMMER	STRASSE
1	15	Talstraße

Hibernate-Log HSQLDB

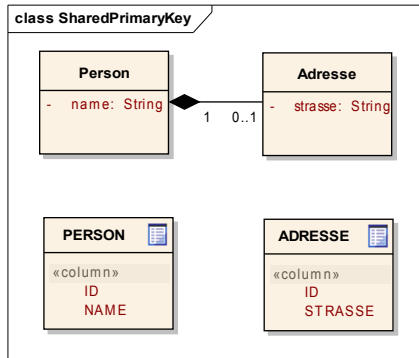
```
insert into Fk_Adresse (id, hausnummer, strasse) values (null, ?, ?)
call identity()
insert into Fk_Person (id, adresse_id, name, vorname) values (null, ?, ?, ?)
call identity()
select fk_person0_id as id14_1_, fk_person0_adresse_id as adresse4_14_1_,
```

tst.one2one.foreignkey

1:1 Beziehung

Shared Primary Key

@PrimaryKeyJoinColumn



Adresse

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

Ein spezieller Hibernate Generator, der sich den Wert des Primärschlüssels von der Adresse holt.

Person

```
@Entity
@GeneratedValue(name = "foreignGenerator", strategy = "foreign",
    parameters = { @Parameter(name = "property", value = "adresse") })

public class SPK_Person {
    private Long id;

    @Id
    @GeneratedValue(generator="foreignGenerator")
    public Long getId() {
        return id;
    }

    private SPK_Adresse adresse;

    @OneToOne(cascade=CascadeType.ALL)
    public SPK_Adresse getAdresse() {
        return adresse;
    }
    [...]
}
```



Hier wird zur Belegung des Primärschlüssels der zuvor definierte Generator benutzt.

tst.one2one.sharedprimarykey

1:1 Beziehung

Shared Primary Key (Fortsetzung)

@PrimaryKeyJoinColumn

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

SPK_Person person = new SPK_Person();
person.setVorname("Donald");
person.setName("Duck");

SPK_Adresse adresse = new SPK_Adresse();
adresse.setStrasse("Talstraße");
adresse.setHausnummer("15");

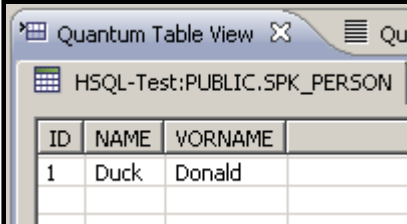
person.setAdresse(adresse);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();

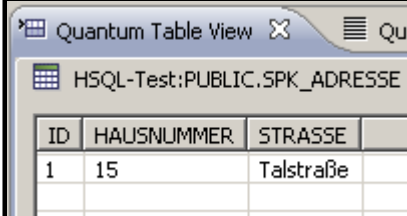
em.clear();

SPK_Person
    loadedPerson = em.find(SPK_Person.class, person.getId());

System.out.println(person.getAdresse().getStrasse());
```



ID	NAME	VORNAME
1	Duck	Donald



ID	HAUSNUMMER	STRASSE
1	15	Talstraße

Hibernate-Log HSQLDB

```
insert into SPK_Adresse (id, strasse, ...
call identity()
insert into SPK_Person (id, name, ...
select spk_person0_id ... on spk_person0_.adresse_id=spk_adress1_.id where spk_person0_.id=?
```

call identity() wird
nur einmal gerufen!

tst.one2one.sharedprimarykey

1:1 Beziehung unidirektional

Delete Adresse – naiver Fehlversuch

Delete Adresse

Testcode – Delete Adresse

```
EntityManager em = JpaUtil.getEntityManager();

em.getTransaction().begin();

// Person anhand seines Primärschlüssels laden.
Fk_Person person = em.find(Fk_Person.class, pk);

person.setAdresse(null);

// Person wieder speichern
em.getTransaction().commit();
```

Hibernate-Log

```
select fk_person0_id as id2_1, ...
update Fk_Person set adresse_fk=null, ...
```

ID	NAME	VORNAME	ADRESSE_FK
1	Duck	Donald	<NULL>

ID	HAUSNUMMER	STRASSE
1	15	Talstraße

- Das Objekt person verliert wie gewünscht die Adresse.
- Aber die Adresse bleibt weiterhin gespeichert.
- Lediglich der Fremdschlüssel auf die Adresse wird auf NULL gesetzt.

tst.one2one.foreignkey

1:1 Beziehung unidirektional

Delete Adresse – jetzt richtig

Delete Adresse

Testcode – Delete Adresse

```
EntityManager em = JpaUtil.getEntityManager();  
em.getTransaction().begin();  
  
// Person anhand seines Primärschlüssels laden.  
Fk_Person person = em.find(Fk_Person.class, pk);  
  
em.remove(person.getAdresse());  
  
person.setAdresse(null);  
  
// Person wieder speichern  
em.getTransaction().commit();
```

- Für eine korrekte Funktion müssen die Operationen mit dem EntityManager und die Beziehungen der Java-Objekte konsistent sein.

Hibernate-Log

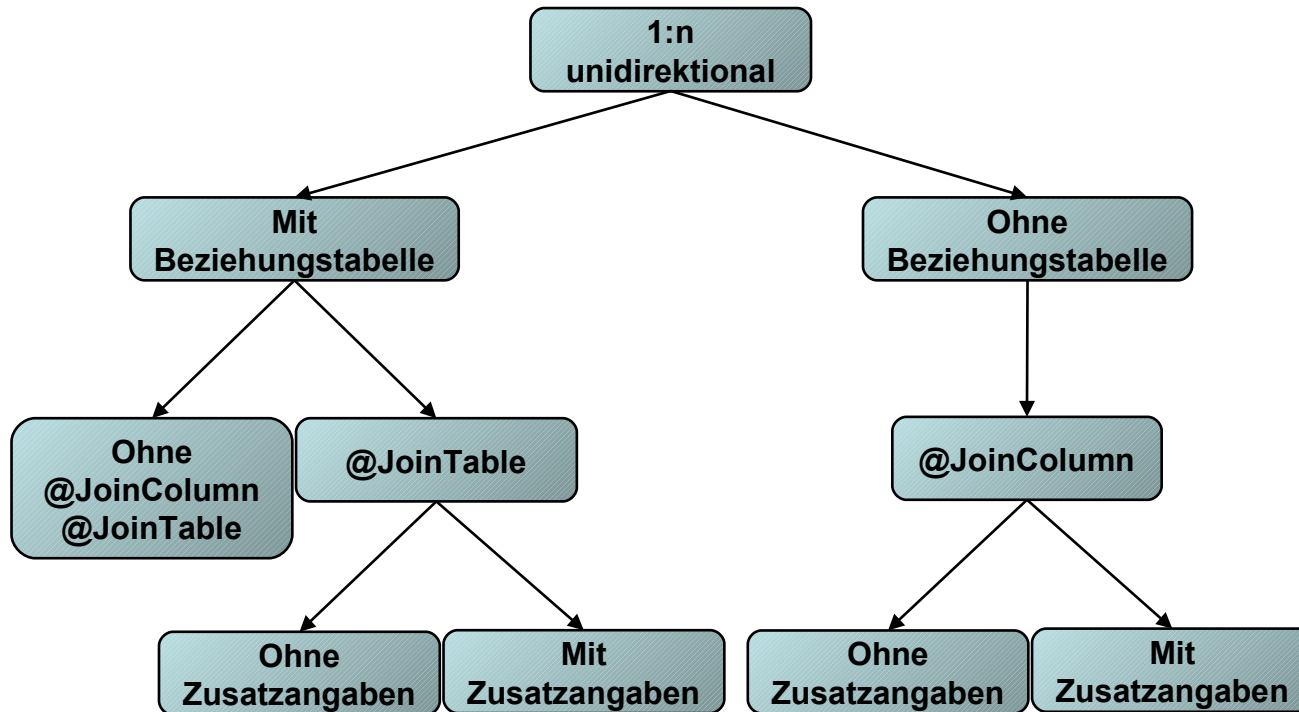
```
select fk_person0_.id as id2_1_, ...  
update Fk_Person set adresse_fk=?, ...  
delete from Fk_Adresse where id=?
```

tst.one2one.foreignkey

1:n Beziehung unidirektional

Varianten

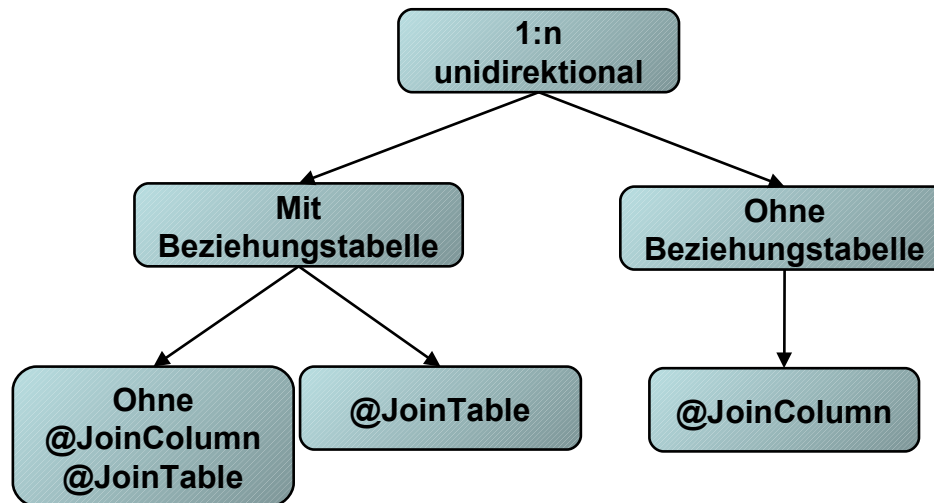
Varianten



1:n Beziehung unidirektional

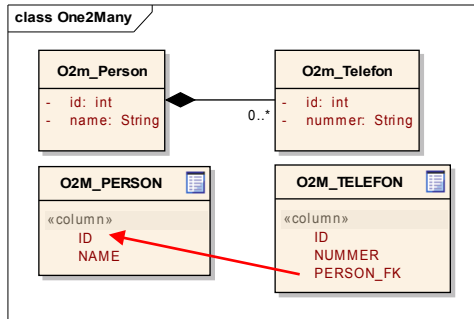
Varianten

Varianten



1:n Beziehung unidirektional ohne zusätzlicher Mappingtabelle

@OneToMany
@JoinColumn



Person

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "person_fk")
public List<O2m_Telefon> getTelefonNummern() {
    if (telefonNummern == null) {
        telefonNummern = new ArrayList<O2m_Telefon>();
    }

    return telefonNummern;
}
```

Hier angeben, dass es sich um eine 1:n-Beziehung handelt.

Telefon

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

Testcode - Insert

```
EntityManager em = JpaUtil.getEntityManager();

O2m_Person person = new O2m_Person();
person.setVorname("Daisy");
person.setName("Duck");

O2m_Telefon telefon = new O2m_Telefon();
telefon.setNummer("0123-23456");
telefon.setBemerkung("privat");
person.getTelefonNummern().add(telefon);

telefon = new O2m_Telefon();
telefon.setNummer("0177-23456");
telefon.setBemerkung("mobil");
person.getTelefonNummern().add(telefon);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
```

Spalte für den Fremdschlüssel. Ohne @JoinColumn wird automatisch eine separate Tabelle für die Beziehung benutzt.

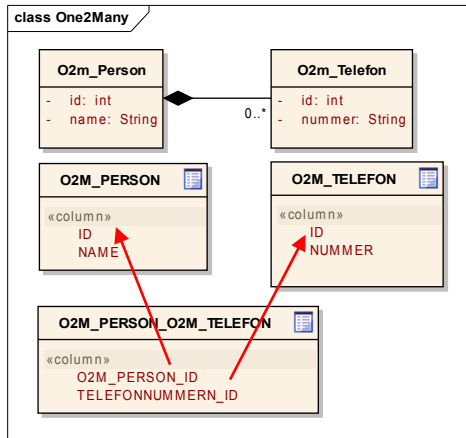
Hibernate-Log HSQLDB

```
insert into O2m_Person (id, name, ...)
call identity()
insert into O2m_Telefon (bemerkung, ...)
insert into O2m_Telefon (bemerkung, ...)
update O2m_Telefon set person_fk=? where nummer=?
update O2m_Telefon set person_fk=? where nummer=?
```

tst.one2many

1:n Beziehung unidirektional mit zusätzlicher Beziehungstabelle

@OneToMany
@JoinColumn



Person

```
@OneToMany(cascade = CascadeType.ALL)
public List<O2m_Telefon> getTelefonNummern() {
    if (telefonNummern == null) {
        telefonNummern = new ArrayList<O2m_Telefon>();
    }

    return telefonNummern;
}
```

Hier angeben, dass es sich um eine 1:n-Beziehung handelt.

Telefon

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

Testcode - Insert

```
EntityManager em = JpaUtil.getEntityManager();

O2m_Person person = new O2m_Person();
person.setVorname("Daisy");
person.setName("Duck");

O2m_Telefon telefon = new O2m_Telefon();
telefon.setNummer("0123-23456");
telefon.setBemerkung("privat");
person.getTelefonNummern().add(telefon);

telefon = new O2m_Telefon();
telefon.setNummer("0177-23456");
telefon.setBemerkung("mobil");
person.getTelefonNummern().add(telefon);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
```

Hibernate-Log

```
insert into O2m_Person (id, name, vorname) values ...
call identity()
insert into O2m_Telefon (id, nummer, bemerkung) ...
call identity()
insert into O2m_Telefon (id, nummer, bemerkung) ...
call identity()
insert into O2m_Person_O2m_Telefon (O2m_Person_id, ...
insert into O2m_Person_O2m_Telefon (O2m_Person_id, ...
```

tst.one2many

1:n Beziehung unidirektional

Delete Person

Delete Person

Testcode – Delete Person

```
EntityManager em = JpaUtil.getEntityManager();
em.getTransaction().begin();

person = em.merge(person);
em.remove(person);

em.getTransaction().commit();
```

Das detached Objekt person dem neuen EntityManager übergeben.

Das Objekt person löschen.

Hibernate-Log

```
select o2m_person0_id as id17_1_, o2m_person0_name ...
update O2m_Telefon set myperson=null where myperson=?
delete from O2m_Telefon where id=?
delete from O2m_Telefon where id=?
delete from O2m_Person where id=?
```

- Alternativ zum Aufruf von `em.merge` könnte das Objekt mittels `em.find` ohne anhängende Telefonnummern neu geladen werden. Was allerdings für `em.remove` zu einem zusätzlichen Select-Statement im Vergleich zu `em.merge` führt.
- Sofern wie im Beispiel das transitive Remove aktiv ist, werden bei Bedarf alle Telefonnummern nachgeladen und einzeln gelöscht.
- Ist das transitive Remove nicht aktiv, so wird lediglich der Datensatz in `O2M_PERSON` gelöscht und alle Fremdschlüssel der dazugehörigen Telefonnummern in `O2M_TELEFON` auf `NULL` gesetzt.

tst.one2many

1:n Beziehung unidirektional

Delete Telefon – 1. naiver Fehlversuch

Delete Telefon

Testcode – Delete Telefon

```
EntityManager em = JpaUtil.getEntityManager();
em.getTransaction().begin();

person = em.merge(person);
List telefonNummern = person.getTelefonNummern();

// Einfach eine Telefonnummer entfernen
telefonNummern.remove(0);

// person wieder speichern
em.persist(person);
em.getTransaction().commit();
```

Das detached Objekt person dem neuen EntityManager übergeben.

- Das Objekt person verliert wie gewünscht die Telefonnummer.
- Aber die Telefonnummer bleibt weiterhin mit NULL als Fremdschlüssel gespeichert.

ID	BEMERKUNG	NUMMER	MYPERSON
1	privat	07152-23456	<NULL>
2	mobil	0177-23456	1

Hibernate-Log

```
select o2m_person0_id as id17_0_, ...
select telefonnum0_myperson as myperson1_, ...
update O2m_Telefon set myperson=null where myperson=? and id=?
```

tst.one2many

1:n Beziehung unidirektional

Delete Telefon – 2. naiver Fehlversuch

Delete Telefon

Testcode – Delete Telefon

```
EntityManager em = JpaUtil.getEntityManager();
em.getTransaction().begin();

person = em.merge(person);
List<O2m_Telefon>
    telefonNummern = person.getTelefonNummern();

// Einfach eine Telefonnummer aus der Liste löschen
em.remove(telefonNummern.get(0));

em.getTransaction().commit();
```

Das detached Objekt person dem neuen EntityManager übergeben.

- Der Versuch wird mit einer Exception quittiert. Diese tritt auf, weil Hibernate versucht, das geänderte Objekt person zu speichern.
- Die Exception bemängelt den Versuch eine gelöschte Entität zu speichern.
- Ursache für die Exception ist die Telefonnummer, welche mit em.remove gelöscht wurde, aber immer noch als Objekt in person enthalten ist.

Hibernate-Log

```
select o2m_person0_id as id17_0_, ...
select telefonnum0_myperson as myperson1_, ...
Exception in thread "main" javax.persistence.RollbackException:
Error while committing the transaction
at ...
Caused by: org.hibernate.ObjectDeletedException:
deleted entity passed to persist: [tst.one2many.O2m_Telefon#<null>]
at ...
```

tst.one2many

1:n Beziehung unidirektional

Delete Telefon

Delete Telefon

Testcode – Delete Telefon

```
EntityManager em = JpaUtil.getEntityManager();
em.getTransaction().begin();

person = em.merge(person);
List<O2m_Telefon>
    telefonNummern = person.getTelefonNummern();

// Einfach eine Telefonnummer löschen und aus der
// Liste der Telefonnummern entfernen.
em.remove(telefonNummern.get(0));
telefonNummern.remove(0);

em.getTransaction().commit();
```

Das detached Objekt person dem neuen EntityManager übergeben.

- Für eine korrekte Funktion müssen die Operationen mit dem EntityManager und die Beziehungen der Java-Objekte konsistent sein.
- In einem „echten“ Programm wird man den Code zur Pflege der Beziehungen in den Entities oder DAOs unterbringen.
- Für bidirektionale Beziehungen benötigt man zusätzlich noch Code zur Pflege der Rückwärtsreferenz.

Hibernate-Log

```
select o2m_person0_.id as id17_0_, ...
select telefonnum0_.myperson as myperson1_, ...
update O2m_Telefon set myperson=null where myperson=? and id=?
delete from O2m_Telefon where id=?
```

tst.one2many

1:n Beziehung unidirektional

Delete Telefon

Delete Telefon
Delete Orphan



Person – Delete Orphan

```
private List<O2m_Telefon> telefonNummern;  
  
@JoinColumn(name="myperson")  
@OneToMany(cascade = CascadeType.ALL, orphanRemoval=true)  
public List<O2m_Telefon> getTelefonNummern() {  
    [...]  
}
```

Automatisches Löschen
verwaister Objekte, die aus einer
1:n Beziehung entfernt wurden.

Testcode – Delete Telefon Orphan

```
EntityManager em = JpaUtil.getEntityManager();  
em.getTransaction().begin();  
  
person = em.merge(person);  
List<O2m_Telefon>  
    telefonNummern = person.getTelefonNummern();  
  
// Einfach eine Telefonnummer aus der Beziehung  
// entfernen.  
telefonNummern.remove(0);  
  
em.getTransaction().commit();
```

Das detached Objekt person dem
neuen EntityManager übergeben.

Auf Portabilitätsgründen dürfen so entfernte
Objekte keinesfalls wieder neu zugewiesen
werden.

Hibernate-Log

```
select o2m_person0_id as id23_1,...  
update O2m_Telefon set myperson=null where myperson=? and id=?  
delete from O2m_Telefon where id=?
```

tst.one2many

- Strategie bzgl. dem Laden referenzierter Daten
 - Eager: Gleich alles mitladen.
 - Lazy: Erst dann laden, wenn ein Zugriff darauf erfolgt.
- Achtung!
 - Gemäß EJB3 Spezifikation ist derzeit Lazy-Loading optional.
 - Lazy-Loading führt bei Hibernate zu Proxy-Objekten. Probleme mit Objektidentitäten möglich.
 - Lazy-Loading führt ggf. zu Detached Entities und später ggf. zu Exceptions.
 - Die „Tiefe“ des Mitladens ist beschränkt, aber einstellbar.
 - Bei einem merge werden LAZY-Felder, die noch nicht gefetched wurden, auch nicht gemerged.
- Relevante Stellen
 - Annotation bei Objektattributen und Beziehungen
 - Fetchstrategie bei Abfragen
- Defaults
 - Eager bei Objektattributen und 1:1 und n:1 Beziehungen.
 - Lazy bei ?:n Beziehungen

Lazy vs. Eager Loading

Beispiele

@Basic
@OneToMany

Ändern der Strategie
bei Attributen von
EAGER auf LAZY

Ändern der Strategie bei
einer 1:n Beziehung von
LAZY auf EAGER

Person

```
@Lob
// Scheint in Hibernate
// nicht implementiert zu sein
@Basic(fetch = FetchType.LAZY)
public byte[] getFoto() {
    return foto;
}
```

```
@OneToMany(cascade = CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name = "person_fk")
public List<LE_Telefon> getTelefonNummern() {
    if (telefonNummern == null) {
        telefonNummern = new ArrayList<LE_Telefon>();
    }

    return telefonNummern;
}
```

SQL bei LAZY

```
SELECT
  le_person0_.id AS id11_0_,
  le_person0_.name AS name11_0_,
  le_person0_.vorname AS vorname11_0_,
  le_person0_.foto AS foto11_0_
FROM
  LE_Person le_person0_
WHERE
  le_person0_.id=?

SELECT
  telefonnum0_.person_fk AS person3_1_,
  telefonnum0_.nummer AS nummer1_,
  telefonnum0_.nummer AS nummer12_0_,
  telefonnum0_.bemerkung AS bemerkung12_0_
FROM
  LE_Telefon telefonnum0_
WHERE
  telefonnum0_.person_fk=?
```

SQL bei EAGER

```
SELECT
  le_person0_.id AS id11_1_,
  le_person0_.name AS name11_1_,
  le_person0_.vorname AS vorname11_1_,
  le_person0_.foto AS foto11_1_,
  telefonnum1_.person_fk AS person3_3_,
  telefonnum1_.nummer AS nummer3_,
  telefonnum1_.nummer AS nummer12_0_,
  telefonnum1_.bemerkung AS bemerkung12_0_
FROM
  LE_Person le_person0_ LEFT OUTER JOIN LE_Telefon telefonnum1_
ON
  le_person0_.id=telefonnum1_.person_fk
WHERE
  le_person0_.id=?
```

tst.lazyeager

Ergänzend zu unidirektionalen Beziehungen muss man ein wenig mehr tun:

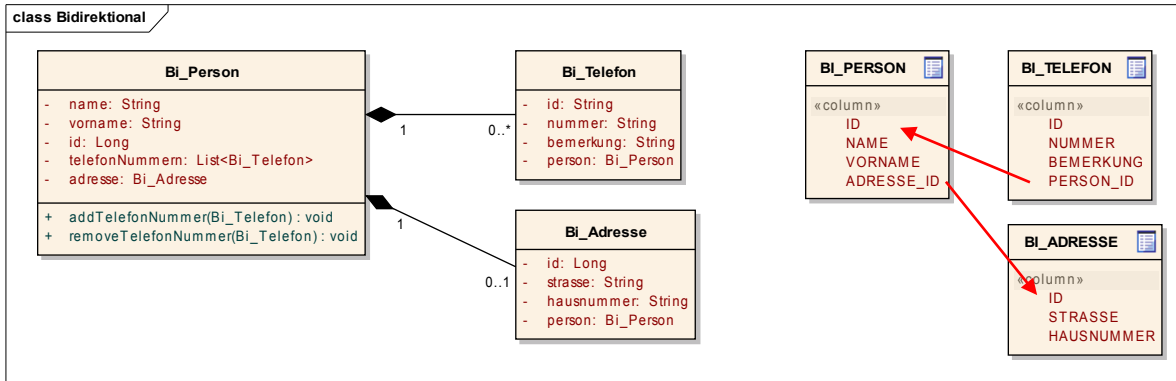
- „mappedBy“ verweist auf die zuständige Seite der Beziehung (owner). Dies erfolgt bei unidirektionalen Beziehungen implizit.
 - Der Owner einer Beziehung legt fest, wo und wie der Fremdschlüssel der Beziehung gespeichert wird. → @JoinColumn, @JoinTable
 - Bei 1:n und n:1 Beziehungen muss immer die n-Seite der owner sein. Damit wird „mappedBy“ immer bei der 1-er Seite notiert.
- Objektreferenzen beider Seiten müssen gepflegt werden.
- Achtung!!! Bei bidirektionalen Beziehungen können aufgrund der Standardladestrategie „EAGER“ der Rückwärtsreferenz ungewollt ganze Objektnetze aus der Datenbank gelesen werden.

Allerdings:

- Lassen sich durch Einbeziehen der Rückwärtsreferenzen Datenabfragen flexibler formulieren.
- Können Objekt-Teilnetze kontrolliert durch Aufruf der Rückwärtsreferenzen nachgeladen werden.

1:1 Beziehung bidirektional

@OneToOne
mappedBy



Person

```
@Entity
public class Bi_Person {
    private Bi_Adresse adresse;

    @OneToOne(cascade = CascadeType.ALL)
    public Bi_Adresse getAdresse() {
        return adresse;
    }

    public void setAdresse(Bi_Adresse adresse) {
        this.adresse = adresse;
    }
    [...]
}
```

Adresse

```
@Entity
public class Bi_Adresse {
    private Bi_Person person;

    @OneToOne(mappedBy="adresse")
    public Bi_Person getPerson() {
        return person;
    }

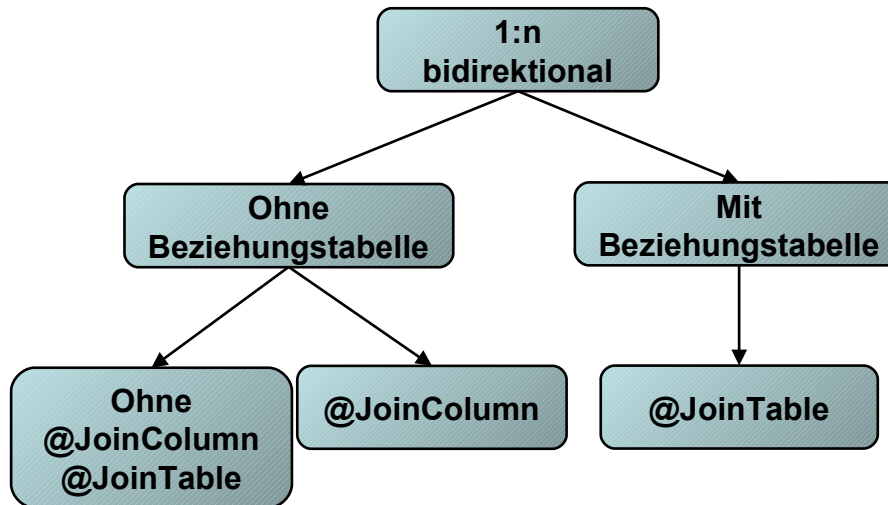
    public void setPerson(Bi_Person person) {
        this.person = person;
    }
    [...]
}
```

tst.bidir

1:n Beziehung bidirektional

Varianten

Varianten



1:n Beziehung bidirektional

@OneToMany
@ManyToOne

Person

```
private List<Bi_Telefon> telefonNummern;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "person")
public List<Bi_Telefon> getTelefonNummern() {
    if (telefonNummern == null) {
        telefonNummern = new ArrayList<Bi_Telefon>();
    }
    return telefonNummern;
}

public void setTelefonNummern(List<Bi_Telefon> nummern) {
    telefonNummern = nummern;
}

public void addTelefonNummer(Bi_Telefon telefon) {
    telefon.setPerson(this);
    getTelefonNummern().add(telefon);
}

public void removeTelefonNummer(Bi_Telefon telefon) {
    if (getTelefonNummern().remove(telefon)) {
        telefon.setPerson(null);
    }
    else {
        throw new IllegalStateException();
    }
}
}
```

Telefon

```
private Bi_Person person;

@ManyToOne
public Bi_Person getPerson() {
    return person;
}

public void setPerson(Bi_Person person) {
    this.person = person;
}
}
```

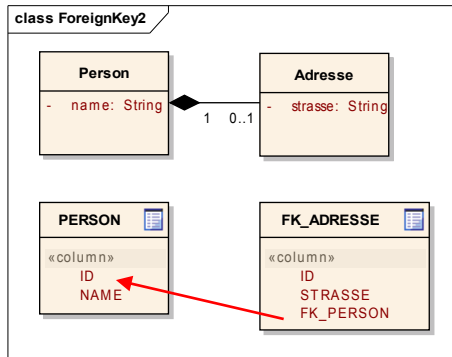
Pflege der Objektreferenzen.

tst.bidir

1:1 Beziehung bidirektional

Foreign Key II

Sonderfall



- Durch die bidirektionale Beziehung ergibt sich die Möglichkeit einer weiteren Variante für 1:1 Beziehungen.
- Fremdschlüssel wird beim Kindobjekt gespeichert
- Da es seitens der Datenbank wie eine 1:n Beziehung aussieht, ist es sinnvoll die 1:1-Beziehung mittels Unique Constraint zu schützen.

Person

```
private Fk2_Adresse adresse;

@OneToOne(cascade = CascadeType.ALL, mappedBy="person")
public Fk2_Adresse getAdresse() {
    return adresse;
}

public void setAdresse(Fk2_Adresse adresse) {
    this.adresse = adresse;
    if (adresse != null) {
        this.adresse.setPerson(this);
    }
}
```

Testcode

```
Fk2_Person person = new Fk2_Person();
person.setName("Duck");

person.setAdresse(new Fk2_Adresse());
person.getAdresse().setStrasse("Burgallee");
em.persist(person);
```

tst.one2one.foreignkeyreverse

Cascade (Transitive Persistenz) (bei falscher Programmierung)

Desasterszenario

Person

```
private List<CA_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL, mappedBy = "person")  
public List<CA_Telefon> getTelefonNummern() {  
    [...]  
}  
  
public void setTelefonNummern(List<CA_Telefon> nummern) {  
    [...]  
}
```

Telefon

```
private CA_Person person;  
  
@ManyToOne(cascade = CascadeType.ALL)  
public CA_Person getPerson() {  
    [...]  
}  
  
public void setPerson(CA_Person person) {  
    [...]  
}
```

Testcode

```
EntityManager em = JpaUtil.getEntityManager();  
  
CA_Person person = em.find(CA_Person.class, id);  
  
em.getTransaction().begin();  
CA_Telefon telefon = person.getTelefonNummern().get(0);  
em.remove(telefon);  
em.getTransaction().commit();
```

Hibernate-Log

```
select ... from CA_Person ... where ...  
select ... from CA_Telefon ... where ...  
  
delete from CA_Telefon where id=?  
delete from CA_Telefon where id=?  
delete from CA_Person where id=?
```

Das falsche Löschen einer einzigen Telefonnummer kombiniert mit einem ungeschickten CascadeType.ALL führt in Folge zur Löschung der Person und der restlichen Telefonnummern der Person.

```
CA_Telefon telefon = person.getTelefonNummern().get(0);  
person.getTelefonNummern().remove(0);  
telefon.setPerson(null);  
em.remove(telefon);
```

tst.cascade



- **@ElementCollection** ergänzt die Möglichkeiten für 1:n-Beziehungen für Collections:
 - primitiven Typen.
 - Embedded-Objekten.
 - Key/Value-Einträge einer HasMap, wobei hier als Value wieder primitive Typen oder Embeddeds in Frage können.
- Die Daten der in der Collection enthaltenen Daten werden in einer separaten Tabelle gespeichert.
- Mittels **@CollectionTable** und **@AttributeOverride** können die Tabelle, die Tabellenspalten und die Fremdschlüsselspalte zum Speichern der in der Collection enthaltenen Objekte selbst bestimmt werden.
- LAZY Loading ist die Standardladestrategie wie bei allen 1:n-Beziehungen.

Genauso wie bei Embeddeds, ist der Life-Cycle der Objekte in Collections an den Life-Cycle des umgebenden Objektes gekoppelt. Das heißt, dass beim Löschen des äußeren Objektes automatisch die in der Collection enthaltenen Objekte gelöscht werden. In der Objektorientierung spricht man bei existenziell gekoppelten Objekten von einer Komposition.

Bei einer Beziehung mit **@OneToMany** sind beide Life-Cycle getrennt.

Element-Collection

Beispiele Primitivtyp

@ElementCollection
List<String>



Person

```
private List<String> telefonNummernStrings;

@ElementCollection
@OrderColumn(name="index") // sofern die Liste geordnet ist
public List<String> getTelefonNummernStrings() {
    [...]
}

public void setTelefonNummernStrings(List<String> nummern) {
    [...]
}
```

ID	NAME	VORNAME
1	Duck	Daisy

EC_PERSON_ID	TELEFONNUMMERNSTRINGS	INDEX
1	0123-23456	0
1	0177-23456	1

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

Ec_Person person = new Ec_Person();
person.setVorname("Daisy");
person.setName("Duck");

person.getTelefonNummernStrings().add("0123-23456");
person.getTelefonNummernStrings().add("0177-23456");

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
```

Hibernate-Log

```
insert into Ec_Person (id, name, vorname)
  values (null, 'Duck', 'Daisy')
call identity()
insert into Ec_Person_telefonNummern
  (Ec_Person_id, telefonNummern_ORDER,
  TelefonNummern)
  values (1, 0, '0123-23456')
insert into Ec_Person_telefonNummern
  (Ec_Person_id, telefonNummern_ORDER,
  TelefonNummern)
  values (1, 1, '0177-23456')
```

Element-Collection

Beispiel Map

@ElementCollection
Map<String, String>

2.0

Person

```
private Map<String, String> telefonNummernMap;  
  
@ElementCollection(targetClass=java.lang.String.class)  
public Map<String, String> getTelefonNummernMap() {  
    [...]  
}  
  
public void setTelefonNummernMap(  
    Map<String, String> nummern) {  
    [...]  
}
```

Mittels @MapKeyColumn kann die Tabellenspalte für 'Key' eingestellt werden; mittels @Column die für 'Value'.

HSQL lokal:PUBLIC.EC_PERSON

ID	NAME	VORNAME
1	Duck	Daisy

HSQL lokal:PUBLIC.EC_PERSON_TELEFONNUMMERNMAP

EC_PERSON_ID	TELEFONNUMMERNMAP	TELEFONNUMMERNMAP_KEY
1	0177-23456	mobil
1	0123-23456	privat

Testcode

```
EntityManager em = JpaUtil.getEntityManager();  
  
Ec_Person person = new Ec_Person();  
person.setVorname("Daisy");  
person.setName("Duck");  
  
person.getTelefonNummernMap().put("privat", "0123-23456");  
person.getTelefonNummernMap().put("mobil", "0177-23456");  
  
em.getTransaction().begin();  
em.persist(person);  
em.getTransaction().commit();
```

Hibernate-Log

```
insert into Ec_Person (id, name, vorname)  
values (null, 'Duck', 'Daisy')  
call identity()  
insert into Ec_Person_telefonNummernMap  
(Ec_Person_id, telefonNummernMap_KEY,  
telefonNummernMap)  
values (1, 'mobil', '0177-23456')  
insert into Ec_Person_telefonNummernMap  
(Ec_Person_id, telefonNummernMap_KEY,  
telefonNummernMap)  
values (1, 'privat', '0123-23456')
```

Element-Collection

Beispiel Embedded, Teil 1

@ElementCollection
Set<Ec_Telefon>



Person

```
private Set<Ec_Telefon> telefonNummern;  
  
@ElementCollection  
public Set<Ec_Telefon>, String> getTelefonNummern() {  
    [...]  
}  
  
public void setTelefonNummern(  
    Set<Ec_Telefon> nummern) {  
    [...]  
}
```

Telefon

```
@Embeddable  
public class Ec_Telefon {  
    private String nummer;  
  
    public String getNummer() {  
        return nummer;  
    }  
  
    public void setNummer(String nummer) {  
        this.nummer = nummer;  
    }  
  
    private String bemerkung;  
  
    public String getBemerkung() {  
        return bemerkung;  
    }  
  
    public void setBemerkung(String bemerkung) {  
        this.bemerkung = bemerkung;  
    }  
}
```

Element-Collection

Beispiel Embedded, Teil 2

@ElementCollection
Set<Ec_Telefon> 

Testcode

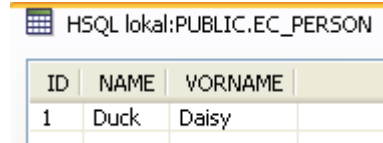
```
EntityManager em = JpaUtil.getEntityManager();

Ec_Person person = new Ec_Person();
person.setVorname("Daisy");
person.setName("Duck");

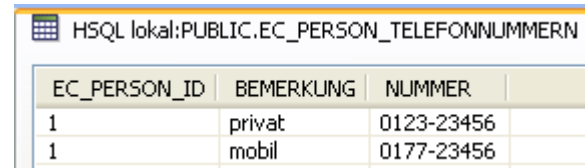
Ec_Telefon telefon = new Ec_Telefon();
telefon.setBemerkung("privat");
telefon.setNummer("0123-23456");
person.getTelefonNummern().add(telefon);

telefon = new Ec_Telefon();
telefon.setBemerkung("mobil");
telefon.setNummer("0177-23456");
person.getTelefonNummern().add(telefon);

em.getTransaction().begin();
em.persist(person);
em.getTransaction().commit();
```



ID	NAME	VORNAME
1	Duck	Daisy



EC_PERSON_ID	BEMERKUNG	NUMMER
1	privat	0123-23456
1	mobil	0177-23456

Hibernate-Log

```
insert into Ec_Person (id, name, vorname)
  values (null, 'Duck', 'Daisy')
call identity()
insert into Ec_Person_telefonNummern
  (Ec_Person_id, bemerkung, nummer)
  values (1, 'mobil', '0177-23456')
insert into Ec_Person_telefonNummern
  (Ec_Person_id, bemerkung, nummer)
  values (1, 'privat', '0123-23456')
```

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

Designempfehlungen

Vererbung

Die JPA bietet drei Möglichkeiten zur Abbildung der Vererbung auf ein Datenmodell:

- SINGLE_TABLE (Standard) → gut

Eine gemeinsame Tabelle je Klassenhierarchie.

- Es ist die performanteste Variante.

- JOINED → gut

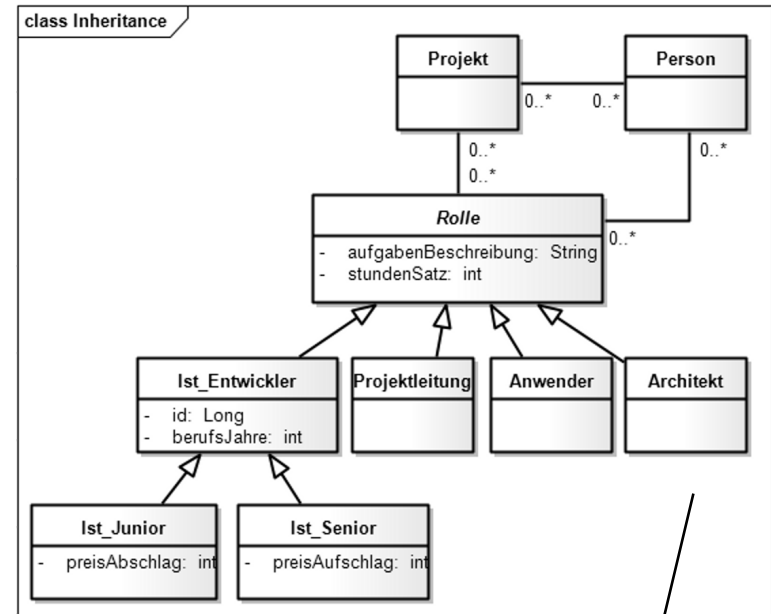
Eine Tabelle für jede einzelne Klasse, unabhängig davon, ob sie abstrakt oder konkret ist.

- Klassen werden eins zu eins abgebildet.
- Platz wird nicht verschwendet.
- Bei Einführung eines neuen Subtyps müssen bestehende Tabellen nicht geändert werden.

- TABLE_PER_CLASS (optional) → schlecht

Eine Tabelle für jede konkrete Klasse.

- Ist lt. JPA 2.0 immer noch optional.
- Schlechte Unterstützung für Polymorphie.
- Abfragen bestehen aus Unions oder mehrere Selects..



In einem Vererbungspfad von beispielsweise 'Rolle' bis 'Junior' kann auf jeder Hierarchieebene die Abbildungsstrategie neu gewählt werden.

Vererbung

SINGLE_TABLE (Standard)

DTYPE

Ist_Entwickler

```
@Entity
@Inheritance
public class Ist_Entwickler {
    [...]
}
```

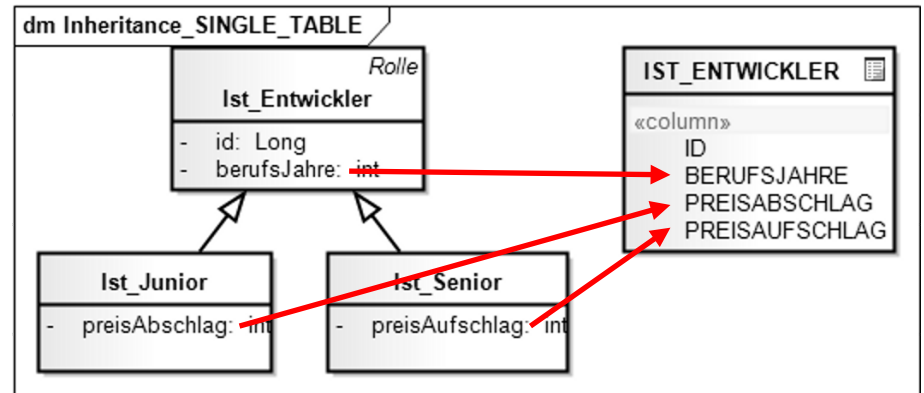
Ist_Junior

```
@Entity
public class Ist_Junior {
    [...]
}
```

Ist_Senior

```
@Entity
public class Ist_Senior {
    [...]
}
```

In 'DTYPE' speichert Hibernate automatisch eine Information zur Erkennung der Ursprungsklasse



ORACLE DDL-Skript

```
create table Ist_Entwickler (
    DTYPE varchar2(31 char) not null,
    id number(19,0) not null,
    berufsJahre number(10,0),
    preisAbschlag number(10,0),
    preisAufschlag number(10,0),
    primary key (id)
);
```

tst.inheritance.singletable

Vererbung SINGLE_TABLE (Standard)

DTYPE
(das Testprogramm)

Testcode

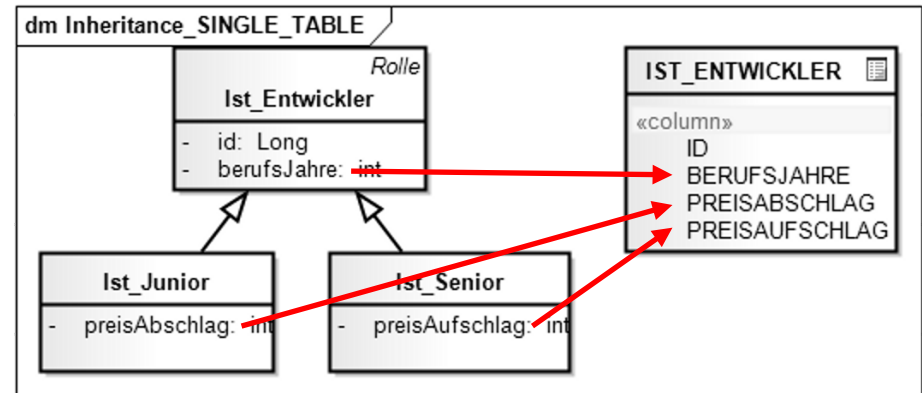
```
em = JpaUtil.getEntityManager();

// Alle Objekte erzeugen
Ist_Entwickler
    entwickler = new Ist_Entwickler(10);

Ist_Junior
    junior = new Ist_Junior(9, 20);

Ist_Senior
    senior = new Ist_Senior(11, 30);

// Alle Objekte in die Datenbank speichern
em.getTransaction().begin();
em.persist(entwickler);
em.persist(junior);
em.persist(senior);
em.getTransaction().commit();
```



HSQldb:PUBLIC.IST_ENTWICKLER

DTYPE	ID	BERUFSJAHRE	PREISABSCHLAG	PREISAUFSCHLAG
Ist_Entwickler	1	10	<NULL>	<NULL>
Ist_Junior	2	9	20	<NULL>
Ist_Senior	3	11	<NULL>	30

Hibernate-Log (ORACLE)

```
select hibernate_sequence.nextval from dual
select hibernate_sequence.nextval from dual
select hibernate_sequence.nextval from dual
insert into Ist_Entwickler (id, berufsJahre, DTYPE) values (1, 10, 'Ist_Entwickler')
insert into Ist_Entwickler (id, berufsJahre, preisAbschlag, DTYPE) values (2, 9, 20, 'Ist_Junior')
insert into Ist_Entwickler (id, berufsJahre, preisAufschlag, DTYPE) values (3, 11, 30, 'Ist_Senior')
```

tst.inheritance.singletable

Vererbung

SINGLE_TABLE (Standard)

@Discriminator...

Ist_Entwickler

```
@Entity
@Inheritance
@DiscriminatorColumn(name = "Diskriminator", discriminatorType = DiscriminatorType.INTEGER)
@DiscriminatorValue("1001")
public class Ist_Entwickler {
    [...]
}
```

Der Name der Diskriminatorkolumne lautet 'Diskriminator', als Diskriminante werden Integer-Werte gespeichert.

Ist_Junior

```
@Entity
@DiscriminatorValue("1002")
public class Ist_Junior {
    [...]
}
```

Ist_Senior

```
@Entity
@DiscriminatorValue("1003")
public class Ist_Senior {
    [...]
}
```

ORACLE DDL-Skript

```
create table Ist_Entwickler (
    Diskriminator number(10,0) not null,
    id number(19,0) not null,
    berufsJahre number(10,0),
    preisAbschlag number(10,0),
    preisAufschlag number(10,0),
    primary key (id)
);
```

Hier sieht man die selbst definierte Spalte für den Diskriminator.

tst.inheritance.singletable

Vererbung

SINGLE_TABLE (Standard)

@Discriminator...
(das Testprogramm)

Testcode

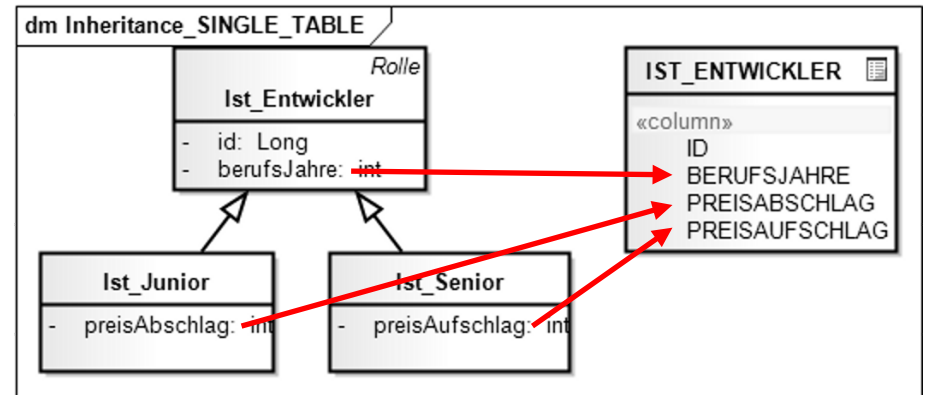
```
em = JpaUtil.getEntityManager();

// Alle Objekte erzeugen
Ist_Entwickler
    entwickler = new Ist_Entwickler(10);

Ist_Junior
    junior = new Ist_Junior(9, 20);

Ist_Senior
    senior = new Ist_Senior(11, 30);

// Alle Objekte in die Datenbank speichern
em.getTransaction().begin();
em.persist(entwickler);
em.persist(junior);
em.persist(senior);
em.getTransaction().commit();
```



HSQldb:PUBLIC.IST_ENTWICKLER

DISKRIMINATOR	ID	BERUFSJAHRE	PREISABSCHLAG	PREISAUFSCHLAG
1001	1	10	<NULL>	<NULL>
1002	2	9	20	<NULL>
1003	3	11	<NULL>	30

Hibernate-Log (ORACLE)

```
select hibernate_sequence.nextval from dual
select hibernate_sequence.nextval from dual
select hibernate_sequence.nextval from dual
insert into Ist_Entwickler (id, berufsJahre, DTYPE) values (1, 10, 'Ist_Entwickler')
insert into Ist_Entwickler (id, berufsJahre, preisAbschlag, DTYPE) values (2, 9, 20, 'Ist_Junior')
insert into Ist_Entwickler (id, berufsJahre, preisAufschlag, DTYPE) values (3, 11, 30, 'Ist_Senior')
```

tst.inheritance.singletable

Vererbung

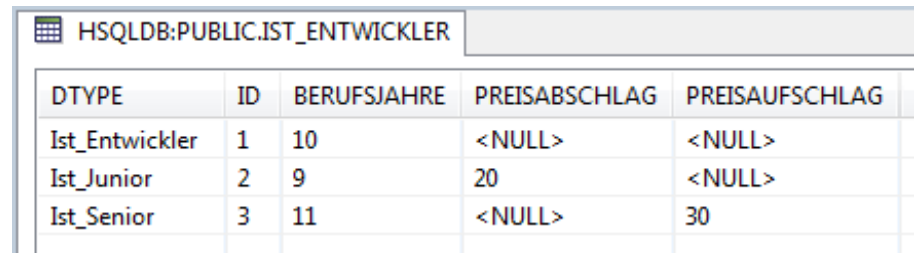
SINGLE_TABLE (Standard)

Polymorphe
Datenabfrage

Testcode

```
Query query = em.createQuery("select e from Ist_Entwickler e where e.berufsJahre >= 10");  
List<Ist_Entwickler> resultList = query.getResultList();  
for (Ist_Entwickler ist_Entwickler : resultList) {  
    System.out.println(ist_Entwickler);  
}
```

Abfragen sind automatisch polymorph. Das heißt, dass bei Abfragen nach Ist_Entwickler automatisch alle Unterklassen in die Abfrage mit einbezogen werden.



DTYPE	ID	BERUFSJAHRE	PREISABSCHLAG	PREISAUFSCHLAG
Ist_Entwickler	1	10	<NULL>	<NULL>
Ist_Junior	2	9	20	<NULL>
Ist_Senior	3	11	<NULL>	30

Hibernate-Log

```
select ist_entwic0_.id as id10_, ist_entwic0_.berufsJahre as berufsJa3_10_,  
       ist_entwic0_.preisAbschlag as preisAbs4_10_, ist_entwic0_.preisAufschlag as preisAuf5_10_,  
       ist_entwic0_.Diskriminator as Diskrimil_10_  
from Ist_Entwickler ist_entwic0_  
where ist_entwic0_.berufsJahre >= '10'  
  
tst.inheritance.singletable.Ist_Entwickler [id=1,berufsJahre=10]  
tst.inheritance.singletable.Ist_Senior [id=3,berufsJahre=11, preisAufschlag=30]
```

tst.inheritance.singletable

Vererbung

SINGLE_TABLE (Standard)

Datenabfrage nach konkreter Entity 

Testcode

```
Query query = em.createQuery("select e from Ist_Entwickler e where type(e) in (Ist_Entwickler) and  
and e.berufsJahre >= 10");  
  
List<Ist_Entwickler> resultList = query.getResultList();  
for (Ist_Entwickler ist_Entwickler : resultList) {  
    System.out.println(ist_Entwickler);  
}
```

Zusätzlich ginge auch:
... where type(e) <> Ist_Junior ..

Liefert alle Datenobjekte, die nicht vom Typ
'Ist_Junior' sind.

HSQLDB:PUBLIC.IST_ENTWICKLER				
DTYPE	ID	BERUFSJAHRE	PREISABSCHLAG	PREISAUFSCHLAG
Ist_Entwickler	1	10	<NULL>	<NULL>
Ist_Junior	2	9	20	<NULL>
Ist_Senior	3	11	<NULL>	30

Hibernate-Log

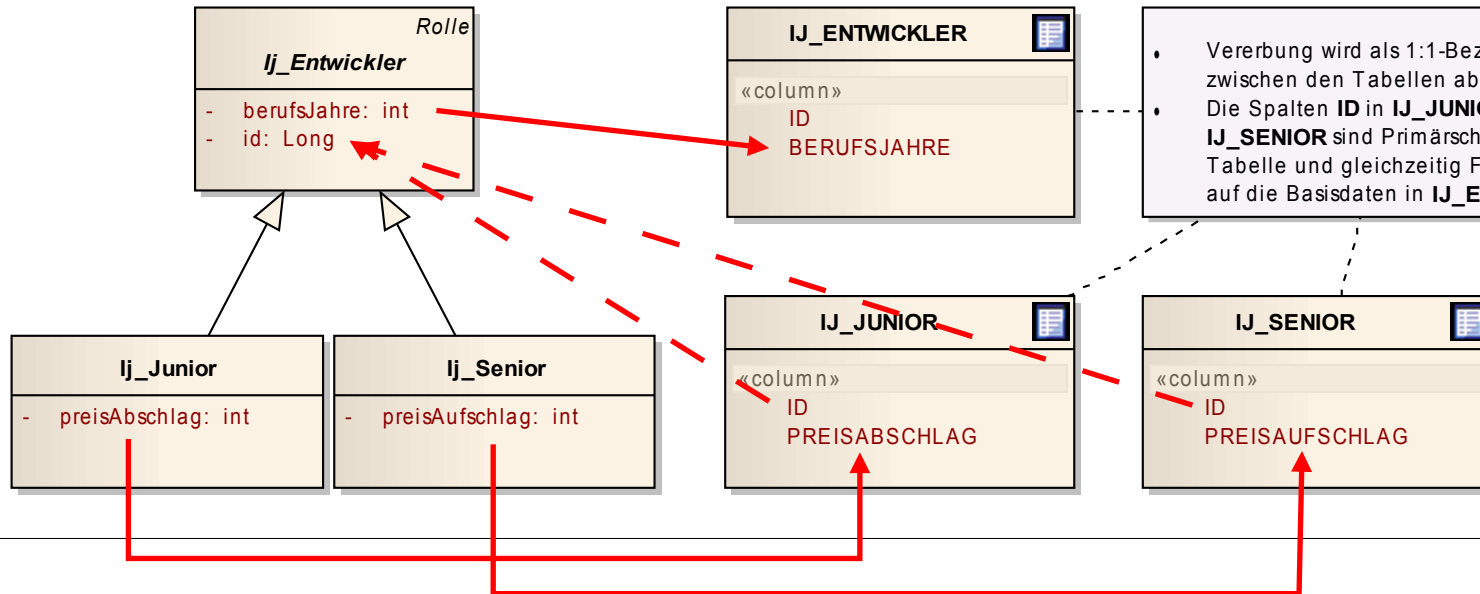
```
select ist_entwic0_.id as id10_, ist_entwic0_.berufsJahre as berufsJa3_10_,  
ist_entwic0_.preisAbschlag as preisAbs4_10_, ist_entwic0_.preisAufschlag as preisAuf5_10_,  
ist_entwic0_.DTYPE as DTYPE10_  
from Ist_Entwickler ist_entwic0_  
where ist_entwic0_.DTYPE='Ist_Entwickler' and ist_entwic0_.berufsJahre >= 10  
  
tst.inheritance.singletable.Ist_Entwickler [id=1,berufsJahre=10]
```

tst.inheritance.singletable

Vererbung JOINED

JOINED
(Modell)

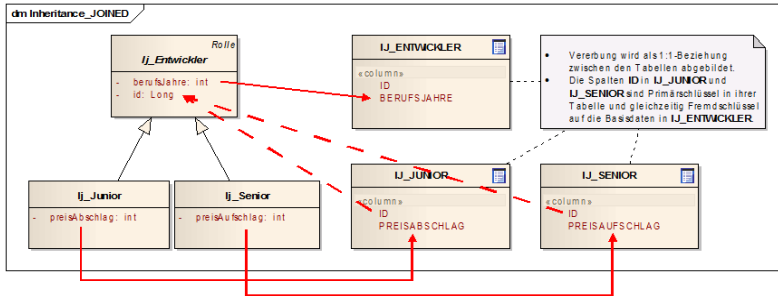
dm Inheritance_JOINED



• Vererbung wird als 1:1-Beziehung zwischen den Tabellen abgebildet.
• Die Spalten **ID** in **IJ_JUNIOR** und **IJ_SENIOR** sind Primärschlüssel in ihrer Tabelle und gleichzeitig Fremdschlüssel auf die Basisdaten in **IJ_ENTWICKLER**.

Vererbung JOINED

JOINED
(DDL)



Ist_Entwickler

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Ij_Entwickler {
    [...]
}
```

Ist_Junior

```
@Entity
public class Ist_Junior {
    [...]
}
```

Ist_Senior

```
@Entity
public class Ist_Senior {
    [...]
}
```

ORACLE DDL-Skript

```
create table Ij_Entwickler (
    id number(19,0) not null,
    berufsJahre number(10,0),
    primary key (id)
);

create table Ij_Junior (
    preisAbschlag number(10,0),
    id number(19,0) not null,
    primary key (id)
);

create table Ij_Senior (
    preisAufschlag number(10,0),
    id number(19,0) not null,
    primary key (id)
);

alter table Ij_Junior
    add constraint FK711B2D8758B6285
    foreign key (id)
    references Ij_Entwickler;

alter table Ij_Senior
    add constraint FK7F95530E58B6285
    foreign key (id)
    references Ij_Entwickler;
```

tst.inheritance.joined

Vererbung JOINED

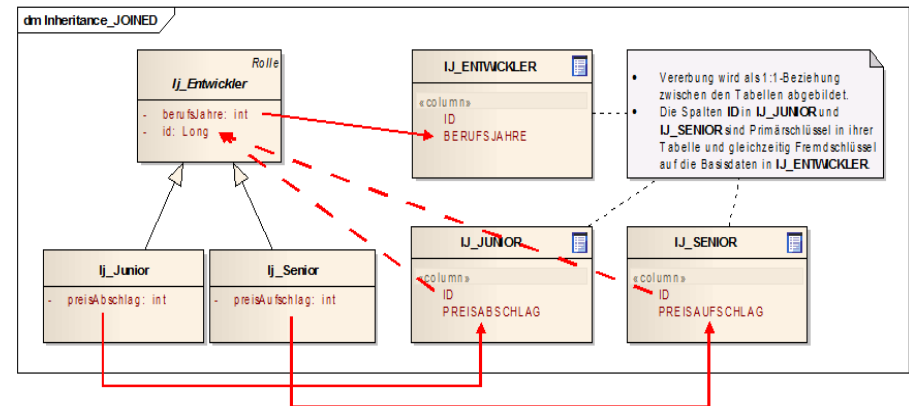
JOINED (das Testprogramm)

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

// Alle Objekte erzeugen
Ij_Junior junior = new Ij_Junior(9, 20);
Ij_Senior senior = new Ij_Senior(11, 30);

// Alle Objekte in die Datenbank speichern
em.getTransaction().begin();
em.persist(junior);
em.persist(senior);
em.getTransaction().commit();
```



Hibernate-Log (ORACLE)

```
select hibernate_sequence.nextval from dual
select hibernate_sequence.nextval from dual
insert into Ij_Entwickler (id, berufsJahre) values (1, 9)
insert into Ij_Junior (id, preisAbschlag) values (1, 20)
insert into Ij_Entwickler (id, berufsJahre) values (2, 11)
insert into Ij_Senior (id, preisAufschlag) values (2, 30)
```

ID	BERUFSJAHRE
1	9
2	11

PREISABSCHLAG	ID
20	1

PREISAUFSCHLAG	ID
30	2

tst.inheritance.joined

Testcode

```
Query query = em.createQuery("select e from Ij_Entwickler e where e.berufsJahre >= 10");  
List<Ij_Entwickler> resultList = query.getResultList();  
for (Ij_Entwickler ist_Entwickler : resultList) {  
    System.out.println(ist_Entwickler);  
}
```

Hibernate-Log (ORACLE)

```
select ij_entwick0_.id as id20_,  
       ij_entwick0_.berufsJahre as berufsJa2_20_,  
       ij_entwick0_1_.preisAufschlag as preisAuf1_28_,  
       ij_entwick0_2_.preisAbschlag as preisAbs1_42_,  
       case when ij_entwick0_.id is not null then 0  
            when ij_entwick0_1_.id is not null then 1  
            when ij_entwick0_2_.id is not null then 2  
       end as clazz_  
from Ij_Entwickler ij_entwick0_  
  left outer join Ij_Senior ij_entwick0_1_  
    on ij_entwick0_.id=ij_entwick0_1_.id  
  left outer join Ij_Junior ij_entwick0_2_  
    on ij_entwick0_.id=ij_entwick0_2_.id  
where ij_entwick0_.berufsJahre>=10  
  
tst.inheritance.joined.Ij_Senior@1fd25ce  
[id=2,preisAufschlag=30,berufsJahre=11]
```

ID	BERUFSJAHRE
1	9
2	11

PREISABSCHLAG	ID
20	1

PREISAUFSCHLAG	ID
30	2

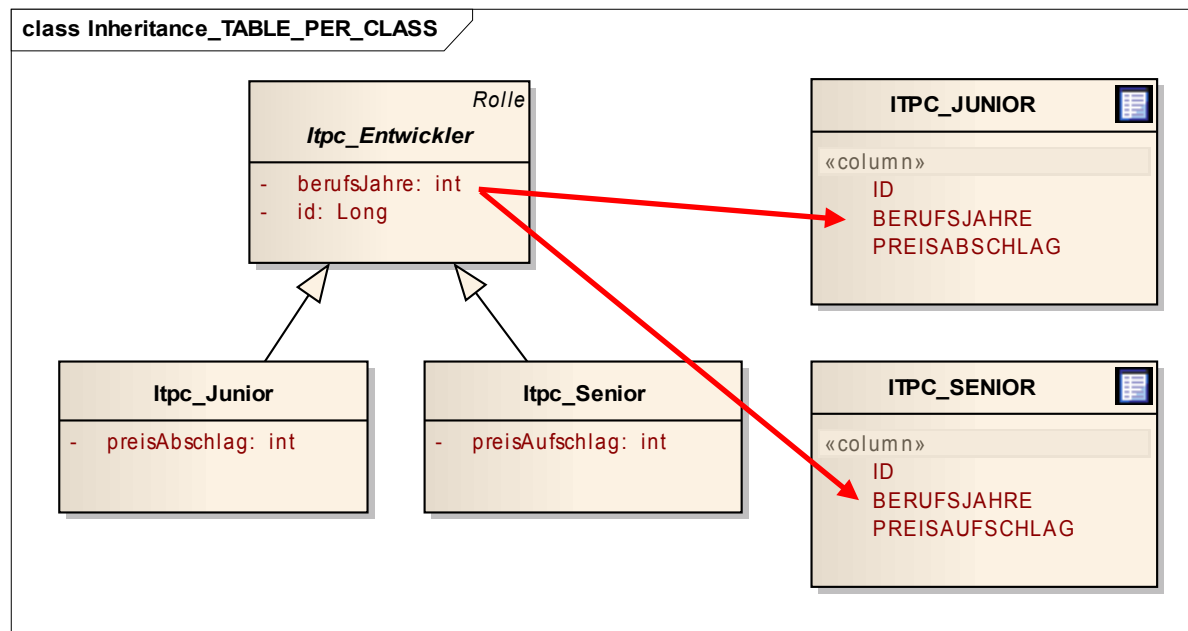
Definiert die Hilfsspalte 'clazz_' in der Select-Clause, um mithilfe der unterschiedlichen case when-Fälle im Zusammenspiel mit den left outer joins einen Diskriminator für die Ursprungsklasse zu bilden. Beim Auslesen des JDBC-ResultSets kann Hibernate anhand des Wertes in 'class_' ein Objekt der betreffenden Ursprungsklasse erzeugen: '1' → Ij_Senior.

tst.inheritance.joined

Vererbung

TABLE_PER_CLASS (optional)

TABLE_PER_CLASS
(Modell)



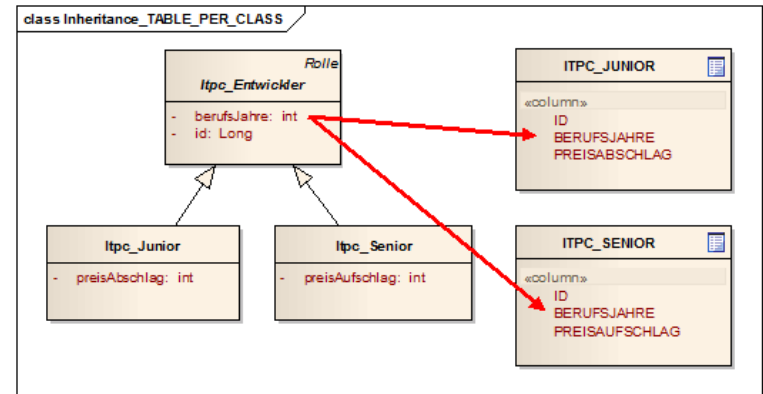
Vererbung

TABLE_PER_CLASS (optional)

TABLE_PER_CLASS
(DDL)

Ist_Entwickler

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@MappedSuperclass
public abstract class Itpc_Entwickler {
    [...]
}
```



Ist_Junior

```
@Entity
public class Ist_Junior {
    [...]
}
```

Ist_Senior

```
@Entity
public class Ist_Senior {
    [...]
}
```

Was die Folie nicht zeigt: Hibernate produziert im DDL leider unnötig eine Tabelle für Ist_Entwickler und benutzt sie auch noch in Abfragen.

ORACLE DDL-Skript

```
create table Itpc_Junior (
    id number(19,0) not null,
    berufsJahre number(10,0),
    preisAbschlag number(10,0),
    primary key (id)
);

create table Itpc_Senior (
    id number(19,0) not null,
    berufsJahre number(10,0),
    preisAufschlag number(10,0),
    primary key (id)
);
```

tst.inheritance.tableperclass

Vererbung

TABLE_PER_CLASS (optional)

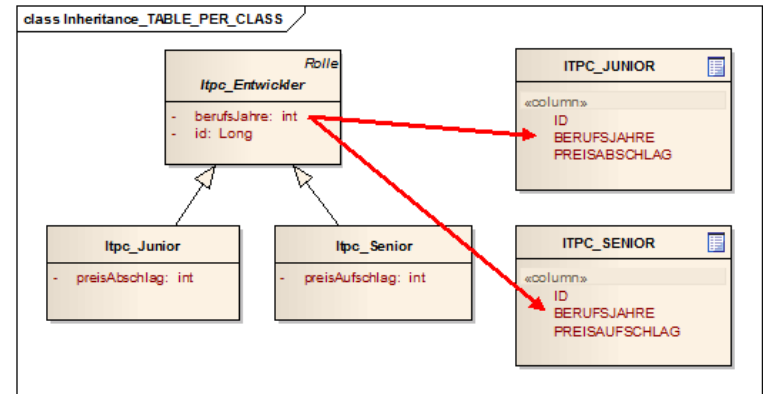
TABLE_PER_CLASS
(das Testprogramm)

Testcode

```
EntityManager em = JpaUtil.getEntityManager();

// Alle Objekte erzeugen
Ij_Junior junior = new Ij_Junior(9, 20);
Ij_Senior senior = new Ij_Senior(11, 30);

// Alle Objekte in die Datenbank speichern
em.getTransaction().begin();
em.persist(junior);
em.persist(senior);
em.getTransaction().commit();
```



Hibernate-Log (ORACLE)

```
select hibernate_sequence.nextval from dual
select hibernate_sequence.nextval from dual
insert into Itpc_Junior (id, berufsJahre, preisAbschlag)
values (1, 9, 20)
insert into Itpc_Senior (id, berufsJahre, preisAufschlag)
values (2, 11, 30)
```

ID	BERUFSJAHRE	PREISABSCHLAG
1	9	20

ID	BERUFSJAHRE	PREISAUFSCHLAG
2	11	30

tst.inheritance.joined

Vererbung

TABLE_PER_CLASS (optional)

Polymorphe
Datenabfrage

Testcode

```
Query query = em.createQuery("select e from Itpc_Entwickler e where e.berufsJahre >= 10");
List<Itpc_Entwickler> resultList = query.getResultList();
for (Itpc_Entwickler ist_Entwickler : resultList) {
    System.out.println(ist_Entwickler);
}
```

Hibernate-Log (ORACLE)

```
select itpc_entwi0_.id as id2_,
       itpc_entwi0_.berufsJahre as berufsJa2_2_
from Itpc_Entwickler itpc_entwi0_
where itpc_entwi0_.berufsJahre>=10

select itpc_junio0_.id as id3_,
       itpc_junio0_.berufsJahre as berufsJa2_3_,
       itpc_junio0_.preisAbschlag as preisAbs3_3_
from Itpc_Junior itpc_junio0_
where itpc_junio0_.berufsJahre>=10

select itpc_senio0_.id as id24_,
       itpc_senio0_.berufsJahre as berufsJa2_24_,
       itpc_senio0_.preisAufschlag as preisAuf3_24_
from Itpc_Senior itpc_senio0_
where itpc_senio0_.berufsJahre>=10

tst.inheritance.joined.Ij_Senior@1fd25ce
[id=2,preisAufschlag=30,berufsJahre=11]
```

Definiert die Hilfsspalte 'clazz_' in der Select-Clause, um mithilfe der unterschiedlichen case when-Fälle im Zusammenspiel mit den left outer joins einen Diskriminator für die Ursprungsklasse zu bilden. Beim Auslesen des JDBC-ResultSets kann Hibernate anhand des Wertes in 'class_' ein Objekt der betreffenden Ursprungsklasse erzeugen: '1' → IjSenior.

tst.inheritance.joined

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

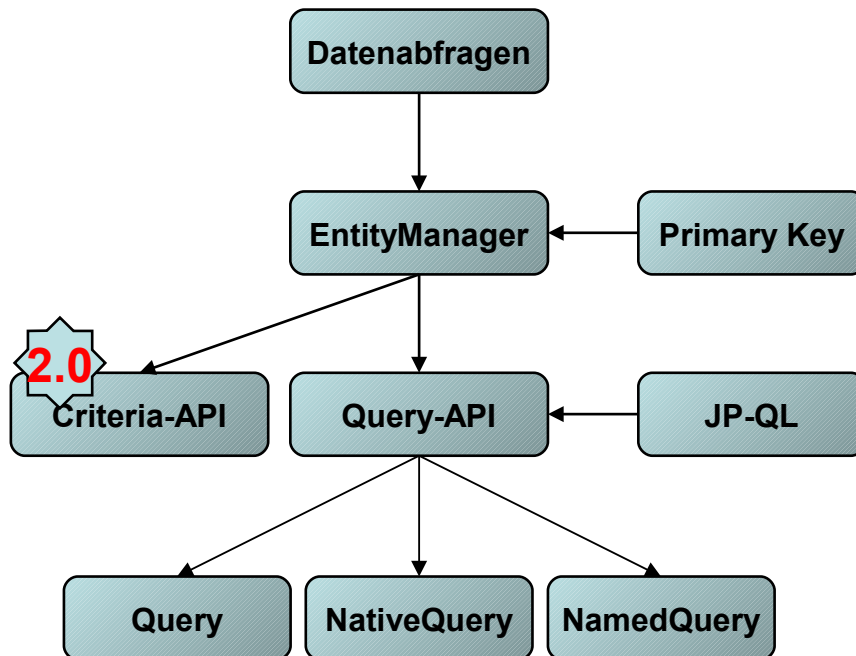
Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

Designempfehlungen



- Erste Anlaufstelle aller Abfragen ist der EntityManager.
- Abfragen anhand des Primärschlüssels unterstützt der EntityManager direkt.
- Alle sonstigen Abfragen erfolgen mittels der Query-API und der Datenabfragesprache JP-QL.
- Die Query-API kennt 3 Querytypen
 - Query: JP-QL Abfrage
 - NativeQuery: Datenbankabhängige SQL-Abfrage
 - NamedQuery: In Annotationen hinterlegte JP-QL Abfrage

Datenabfragen

Via EntityManager / Primärschlüssel

.find(...)
.getReference(...)

.find(...)

```
System.out.println("\n ### Anhand Primärschlüssel vollständig laden");  
person = em.find(Q_Person.class, rememberPrimaryKey);
```

Hibernate-Log

```
### Anhand Primärschlüssel vollständig laden  
select q_person0_id as id11_1_, q_person0_name as name11_1_, ...
```

.getReference(...)

```
System.out.println("\n ### Anhand Primärschlüssel Referenz laden");  
person = em.getReference(Q_Person.class, rememberPrimaryKey);  
System.out.println(" ## Zugriff auf 'name'");  
person.getName();
```

Hibernate-Log

```
## Anhand Primärschlüssel Referenz laden  
## Zugriff auf 'name'  
select q_person0_id as id11_1_, q_person0_name as name11_1_, ...
```

Man beachte, dass der Select erst bei Zugriff auf ein Attribut stattfindet

tst.query

Datenabfragen

Query-API

.createQuery
.setParameter

Erzeugen der verschiedenen Querytypen

```
EntityManager em = JpaUtil.getEntityManager();  
  
Query query = null;  
  
query = em.createQuery(...);  
query = em.createNativeQuery(...);  
query = em.createNamedQuery(...);
```

Parametrisieren einer Query

```
// Nummerierte Parameter  
query.setParameter(nummer, ...);  
  
// Benannte Parameter  
query.setParameter(name, ...);
```

select ... where name=?1

query.setParameter(1, „Duck“);

select ... where name=:param

query.setParameter(„param“, „Duck“);

Die selbe
Positionsnummer bzw.
der selbe Parametername
darf mehrmals in der
Abfrage vorkommen.

Achtung!

Die Spezifikation empfiehlt bei einer
NativeQuery nur Positionsparameter zu
benutzen, um produktneutral zu bleiben.

Datenabfragen

Query-API

```
.executeUpdate()  
.getResultList()  
.getSingleResult()
```

Update oder Delete ausführen

```
int count = query.executeUpdate();
```

Achtung!
Bulk Updates und Deletes wirken sich bei Hibernate nicht auf Datenobjekte in einem optional aktiviertem Second Level Cache aus.

Führt die Query aus und liefert die Anzahl der von der Query aktualisierten bzw. gelöschten Datensätze.
Mit einem Statement können mehrere Datensätze aktualisiert bzw. gelöscht werden.
→ Bulk Updates / Deletes

Select ausführen

```
List list = query.getResultList();
```

```
Object single = query.getSingleResult();
```

Führt die Query aus und liefert eine Liste von Suchergebnissen.

Diese Variante dann verwenden, wenn man genau einen Treffer für die Abfrage erwartet. Sollte die Abfrage dennoch mehr oder weniger als 1 Treffer ergeben, so wird entweder eine `NonUniqueResultException` oder `NoResultException` geworfen.

Steht in der Query irrtümlicher Weise die falsche Abfrageart wie beispielsweise ein Delete anstatt einem Select, dann wird eine `IllegalStateException` geworfen.

Zusammenstellung einer Query mit sequentiellen Statements

```
EntityManager em = JpaUtil.getEntityManager();

Query q = em.createQuery("Select p from Q_Person p
                        where name=:name");

q.setParameter("name", "Duck");
List result = q.getResultList();
```

Zusammenstellung einer Query mittels Method Chaining

```
EntityManager em = JpaUtil.getEntityManager();

List result = em.createQuery("Select p from Q_Person p
                        where name=:name")
                .setParameter("name", "Duck")
                .getResultList();
```

Datenabfragen

Query-API

```
.setFirstResult  
.setMaxResults  
.setFlushMode
```

Einstellungen für die Paginierung

```
query.setFirstResult(startIndex);  
  
query.setMaxResults(count);
```

FlushMode

```
query.setFlushMode(FlushModeType.AUTO);  
  
query.setFlushMode(FlushModeType.COMMIT);
```

FlushModeType.AUTO (default)

Alle Änderungen an Entities im Persistenzkontext werden mit der Datenbank vor der nächsten datenliefernden Abfrage synchronisiert.
Gewährleistet, dass alle Änderungen innerhalb einer Transaktion sich auch in den Abfrageergebnissen niederschlagen.

FlushModeType.COMMIT

Alle Änderungen an Entities im Persistenzkontext werden mit der Datenbank spätestens bis zum Commit synchronisiert.
Ob Änderungen innerhalb einer Transaktion bei folgenden Abfragen berücksichtigt sind, ist nicht vorhersagbar.

Datenabfragen

Query-API

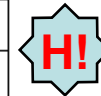
.setHint(...)

Hints (Hinweise)

```
query.setHint(hintName, value);
```

Produktspezifische Hinweise zur Einflussnahme auf die Ausführung der Abfrage. Kennt eine Implementierung einen gesetzten Hinweis nicht, wird er ignoriert.

Hint	Description
org.hibernate.timeout	Query timeout in seconds (eg. new Integer(10))
org.hibernate.fetchSize	Number of rows fetched by the JDBC driver per roundtrip (eg. new Integer(50))
org.hibernate.comment	Add a comment to the SQL query, useful for the DBA (e.g. new String("fetch all orders in 1 statement"))
org.hibernate.cacheable	Whether or not a query is cacheable (eg. new Boolean(true)), defaults to false
org.hibernate.cacheMode	Override the cache mode for this query (eg. CacheMode.REFRESH)
org.hibernate.cacheRegion	Cache region of this query (eg. new String("regionName"))
org.hibernate.readOnly	Entities retrieved by this query will be loaded in a read-only mode where Hibernate will never dirty-check them or make changes persistent (eg. new Boolean(true)), default to false
org.hibernate.flushMode	Flush mode used for this query
org.hibernate.cacheMode	Cache mode used for this query



Achtung!
Nur in Ausnahmefällen verwenden.
Momentan ist jeder Hinweis produktabhängig.

Datenabfragen

Query-API, JP-QL

Select
Left join fetch

Einfacher Select

```
System.out.println("\n ### Einfacher Select");  
Query q = em.createQuery("Select p from Q_Person p where p.name=:name");  
q.setParameter("name", "Kunkel");  
List<Q_Person> resultList = q.getResultList();
```

Hibernate-Log

```
### Einfacher Select  
select q_person0_id as id11_, q_person0_name as name11_, ...  
select telefonnum0_person_fk as person3_1_, ...
```

Man beachte, dass trotz Eager-Loading 2 Selects ausgeführt werden

Fetch Join

```
q = em.createQuery("Select p from Q_Person p left join fetch p.telefonNummern where p.name=:name");
```

Hibernate-Log

```
### Einfacher Select als FETCH JOIN  
select q_person0_id as id11_0_, telefonnum1_nummer as nummer12_1_, ...
```

tst.query

Datenabfragen

Select Count, Paginierung

Select Count
Paginierung

Select Count

```
q = em.createQuery("Select count(t) from Q_Telefon t where t.nummer like :nummer");  
q.setParameter("nummer", "01%");  
Long count = (Long) q.getSingleResult();  
System.out.println(" ## count=" + count);
```

Hibernate-Log

```
### Select count  
select count(q_telefon0_.nummer) as col_0_0_ from Q_Telefon q_telefon0_ where q_telefon0_.nummer like ?  
## count=5
```

Paginierung

```
Query pageQuery = em.createQuery("Select t from Q_Telefon t where t.nummer like :nummer");  
pageQuery.setParameter("nummer", "01%");  
pageQuery.setMaxResults(3);  
pageQuery.setFirstResult(0);  
resultList = pageQuery.getResultList();  
System.out.println(" ## count=" + resultList.size());  
  
pageQuery.setFirstResult(3);  
resultList = pageQuery.getResultList();  
System.out.println(" ## count=" + resultList.size());
```

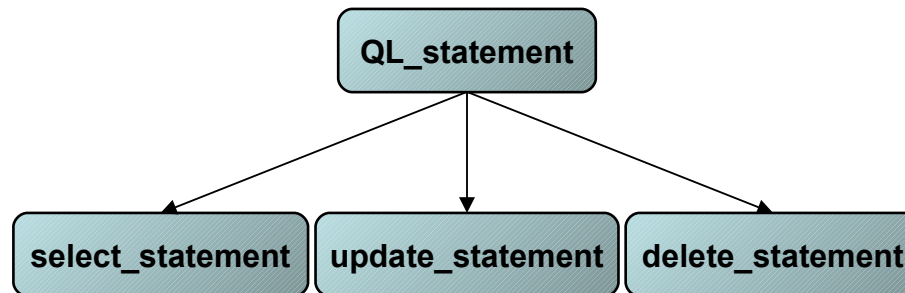
Hibernate-Log

```
### Paginierung  
select top ? q_telefon0_.nummer as nummer12_, q_telefon0_.bemerkung as ...  
## count=3  
select limit ? ? q_telefon0_.nummer as nummer12_, q_telefon0_.bemerkung as ...  
## count=2
```

tst.query

Syntax:

QL_statement ::= select_statement | update_statement | delete_statement



Datenabfragen

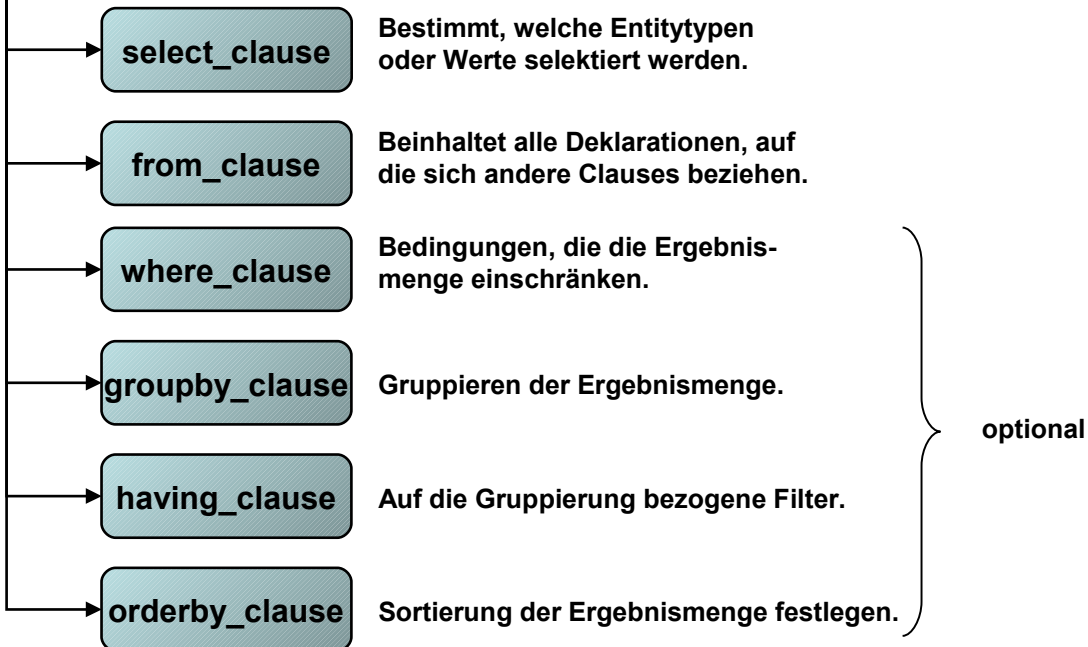
JP-QL - select_statement

select_statement

Syntax:

```
select_statement ::= select_clause from_clause [where_clause]  
[groupby_clause] [having_clause] [orderby_clause]
```

select_statement



Datenabfragen

JP-QL - select_statement

select_clause

Objektselektion I

Selektiert eine Liste von Personen

```
EntityManager em = JpaUtil.getEntityManager();
Query query = em.createQuery("select p from Q_Person p");
List result = query.getResultList();

for (Object object : result) {
    System.out.println(, ### ,, + object);
}
```

Einfachster Datenzugriffspfad
Q_Person mit Alias p.

In der from_clause werden die
notwendigen Datenzugriffspfade
definiert, auf die man sich in der
select_clause bezieht.

Hibernate-Log

```
select q_person0_.id ...
select telefonnum0_.person_fk ...
### tst.query.Q_Person@f37a62[id=1,name=Duck, ...
select telefonnum0_.person_fk ...
### tst.query.Q_Person@e7eec9[id=2,name=Clever, ...
```

tst.query.Q_select_clause

Datenabfragen

JP-QL - select_statement

select_clause

Objektselektion II

Selektiert eine Liste von Personen und Telefonnummern

```
query = em.createQuery("select p, t from Q_Person p, Q_Telefon t");  
result = query.getResultList();
```

```
for (Object object : result) {  
    Object[] objects = (Object[])object;  
    System.out.println("###[0] " + objects[0]);  
    System.out.println("###[1] " + objects[1]);  
}
```

Ein wenig sinnlos; soll lediglich zeigen, dass im Ergebnis mehrere Entitytypen geliefert werden können.

Hibernate-Log

```
select q_person0_.id ... from Q_Person q_person0_, Q_Telefon q_telefon1_  
select telefonnum0_.person_fk ... from Q_Telefon telefonnum0_ where telefonnum0_.person_fk=?  
###[0] tst.query.Q_Person@1497b1[id=1,name=Duck, ...  
###[1] tst.query.Q_Telefon@1f31ad9[nummer=0177-1,bemerkung=mobil]  
###[0] tst.query.Q_Person@1497b1[id=1,name=Duck, ...  
###[1] tst.query.Q_Telefon@167acf2[nummer=0177-2,bemerkung=mobil]  
###[0] tst.query.Q_Person@1497b1[id=1,name=Duck, ...  
###[1] tst.query.Q_Telefon@18b4ccb[nummer=0177-3,bemerkung=mobil]  
###[0] tst.query.Q_Person@1497b1[id=1,name=Duck, ...  
###[1] tst.query.Q_Telefon@5ebac9[nummer=0177-4,bemerkung=mobil]  
###[0] tst.query.Q_Person@1497b1[id=1,name=Duck, ...  
###[1] tst.query.Q_Telefon@138ec91[nummer=0177-5,bemerkung=mobil]  
###[0] tst.query.Q_Person@1497b1[id=1,name=Duck, ...  
###[1] tst.query.Q_Telefon@335053[nummer=07152-23456,bemerkung=privat]  
###[0] tst.query.Q_Person@1497b1[id=1,name=Duck, ...  
###[1] tst.query.Q_Telefon@dea768[nummer=07152-2345667,bemerkung=privat]  
select telefonnum0_.person_fk ... from Q_Telefon telefonnum0_ where telefonnum0_.person_fk=?  
###[0] tst.query.Q_Person@1c0cd80[id=2,name=Clever, ...  
###[1] tst.query.Q_Telefon@1f31ad9[nummer=0177-1,bemerkung=mobil]  
...
```

Die zahlreichen Datensätze sind das Ergebnis des Kreuzprodukts von Q_Person und Q_Telefon

Zu jedem Personenobjekt werden die dazu gehörenden Telefonnummern geladen.

tst.query.Q_select_clause

Datenabfragen

JP-QL - select_statement

select_clause
Diskrete Werte I

Selektiert Name und Vorname der Personen

```
Query query = em.createQuery("select p.name, p.vorname from Q_Person p");  
List result = query.getResultList();  
  
for (Object object : result) {  
    Object[] strings = (Object[])object;  
    System.out.println(" ###[0] " + strings[0]);  
    System.out.println(" ###[1] " + strings[1]);  
}
```

Hibernate-Log

```
select q_person0_.name ... from Q_Person q_person0_  
###[0] Duck  
###[1] Donald  
###[0] Clever  
###[1] Class
```

tst.query.Q_select_clause

Datenabfragen

JP-QL - select_statement

select_clause
Diskrete Werte II
Funktionen

Selektiert Name und Vorname der Personen und verbindet sie zu einem Text

```
query = em.createQuery("select concat(concat(p.name, ', '), p.vorname) from Q_Person p");  
result = query.getResultList();  
  
for (Object object : result) {  
    System.out.println(" ### " + object);  
}
```

Hibernate-Log

```
select ((q_person0_.name||', ')||q_person0_.vorname) as col_0_0_ from Q_Person q_person0_  
### Duck, Donald  
### Clever, Class
```

tst.query.Q_select_clause

Datenabfragen

JP-QL - select_statement

select_clause
Konstruktor

Beispiel nicht-Entity Klasse für einen Bericht oder Liste

```
public class Q_ReportRow {
    private String name;
    private String vorname;

    public Q_ReportRow(String name, String vorname) {
        this.name = name;
        this.vorname = vorname;
    }

    public String toString() {
        return name + ", " + vorname;
    }
}
```

Erzeugt aus Name und Vorname der Personen Objekte eines nicht-Entity Typs

```
query = em.createQuery('select new tst.query.Q_ReportRow(p.name, p.vorname) from Q_Person p');
result = query.getResultList();

for (Object object : result) {
    System.out.println(" ### " + object);
}
```

Hibernate-Log

```
select q_person0.name as col_0_0_, q_person0.vorname as col_1_0_ from Q_Person q_person0
### Duck, Donald
### Clever, Class
```

tst.query.Q_select_clause

Datenabfragen

JP-QL - select_statement

select_clause

Aggregatfunktionen

Anwendung einer einzigen Aggregatfunktionen auf eine Selektion

```
query = em.createQuery("select sum(p.geburtsjahr) from Q_Person p");  
Long summierteGeburtsjahre = (Long)query.getSingleResult();  
System.out.println("### " + summierteGeburtsjahre);
```

JPA-Aggregatfunktionen

- avg
- count
- max
- min
- sum

Hibernate-Log

```
Hibernate: select sum(q_person0_.geburtsjahr) as col_0_0_ from Q_Person q_person0_  
### 3610
```

Anwendung mehrerer Aggregatfunktionen auf eine Selektion

```
query = em.createQuery("select min(p.geburtsjahr), count(p) from Q_Person p");  
result = query.getResultList();  
  
for (Object object : result) {  
    Object[] objects = (Object[])object;  
    System.out.println("###[0] " + objects[0]);  
    System.out.println("###[1] " + objects[1]);  
}
```

Hibernate-Log

```
select min(q_person0_.geburtsjahr) as col_0_0_, count(q_person0_.id) as col_1_0_ from Q_Person q_person0_  
###[0] 1798  
###[1] 2
```

tst.query.Q_select_clause

Datenabfragen

JP-QL - select_statement

from_clause

Syntax:

```
from_clause ::= FROM identification_variable_declaration  
{, {identification_variable_declaration | collection_member_declaration}}*
```

- Hier werden alle Datenzugriffspfade mit zugehöriger Variable definiert, die in den anderen Clauses benutzt werden.
→ identification_variable_declaration
- Es muss mindestens ein Datenzugriffspfad definiert sein.
- Die Pfade können beliebig lang geschachtelt sein.
- Verschiedene Join-Typen:
 - Inner Join
 - Outer Join
 - Fetch Join
 - Impliziter Join bei n:1 und 1:1 in der select_clause

Einfachster Datenzugriffspfad: Q_Person

```
SELECT p FROM Q_Person p  
ORDER BY p.name ASC
```

abstract_schema_name: Q_Person
identification_variable: p

Datenabfragen

JP-QL - select_statement

from_clause

identification_variable_declaration
range_variable_declaration

Syntax:

identification_variable_declaration ::= range_variable_declaration { join |
fetch_join }*

range_variable_declaration ::= abstract_schema_name [AS]
identification_variable

- Das Schlüsselwort AS vor dem Variablennamen ist optional
- abstract_schema_name entspricht einfach dem Entityname

- Einfachste Form `SELECT p FROM Q_Person p`

- Ein vielleicht ungeschicktes Fallbeispiel, zeigt aber die mehrfache Verwendung des selben abstract_schema_name → theta-join

```
SELECT p1 FROM Q_Person AS p1, Q_Person AS p2  
WHERE p1.name < p2.name AND p2.vorname = ,Donald'
```

„theta-join“ – Impliziter Join
ohne definierte Objektbeziehung.

Hibernate-Log

```
select ...  
from Q_Person q_person0_, Q_Person q_person1_  
where q_person0_.name < q_person1_.name and q_person1_.name = 'Duck'
```

tst.query.Q_abstract_schema_name

Datenabfragen

JP-QL - select_statement

from_clause

identification_variable_declaration

join

Syntax:

identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*

join ::= join_spec join_association_path_expression [**AS**] identification_variable

fetch_join ::= join_spec **FETCH** join_association_path_expression

join_spec ::= [**LEFT** [**OUTER**] | **INNER**] **JOIN**

- Für einen nicht Fetch Join wird wieder eine Variable (**identification_variable**) vergeben, so dass sich andere Clauses darauf beziehen können.
- Für einen Fetch Join wird keine Variable vergeben. Er dient lediglich als Hinweis, die Beziehung EAGER zu laden.
- Der Datenpfad eines Joins kann ausschließlich auf einer Variable aufbauen, die in der From-Clause definiert ist.
- Der Datenpfad eines Joins kann am Ende ein SingleValue oder eine Collection adressieren. Die Zwischenstationen im Datenpfad können nur SingleValues sein. Man kann quasi nicht über eine Collection „hinweggehen“.

Datenabfragen

JP-QL - select_statement

from_clause
Inner Join

Syntax:

identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*

join ::= [**INNER**] **JOIN** join_association_path_expression [**AS**] identification_variable

- Die Schlüsselwörter „INNER“ und „AS“ sind optional.

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategien

- Die Anzahl und die Art der DB-Statements variiert abhängig von der Fetchstrategie.
- Der Zeitpunkt, an dem Daten von der DB angefordert werden, variiert abhängig von der Fetchstrategie.
- Es folgen Beobachtungen am Beispiel der 1:n Beziehung: Person -> Telefon

Code zur Beobachtung der DB-Statements

```
1 Query query = em.createQuery("select distinct p from FE_Person p where p.name='Duck'");
2 List<FE_Person> personen = query.getResultList();
3 for (FE_Person person : personen) {
4     List<FE_Telefon> telefonNummern = person.getTelefonNummern();
5     for (FE_Telefon telefon : telefonNummern) {
6         System.out.println(telefon.getNummer());
7     }
8 }
```

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategie
LAZY

Relevanter Codeausschnitt (Person)

```
private List<FE_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL)  
@JoinColumn(name = "person_fk")  
public List<FE_Telefon> getTelefonNummern() {  
    if (telefonNummern == null) {  
        telefonNummern = new ArrayList<FE_Telefon>();  
    }  
  
    return telefonNummern;  
}
```

Keine Besonderheit, da LAZY der Default für 1:n Beziehungen ist.

Hibernate-Log

```
4 List<FE_Telefon> telefonNummern = person.getTelefonNummern();  
--> select ... from FE_Person fe_person0_  
    where fe_person0_.name='Duck'  
  
5 for (FE_Telefon telefon : telefonNummern) {  
--> select ... from FE_Telefon telefonnum0_  
    where telefonnum0_.person_fk=1  
    select ... from FE_Telefon telefonnum0_  
    where telefonnum0_.person_fk=2
```

Jeweils beim Zugriff auf die erste Telefonnummer einer Person. „Load On Demand“ für jede einzelne Telefonnummer einen separaten Select, sofern nicht mittels BatchSize was anderes eingestellt wurde.

tst.fetch

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategie
EAGER

Relevanter Codeausschnitt (Person)

```
private List<FE_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,  
@JoinColumn(name = "person_fk")  
public List<FE_Telefon> getTelefonNummern() {  
    if (telefonNummern == null) {  
        telefonNummern = new ArrayList<FE_Telefon>();  
    }  
  
    return telefonNummern;  
}
```

Hibernate-Log

```
4 List<FE_Telefon> telefonNummern = person.getTelefonNummern();  
--> select ... from FE_Person fe_person0_  
    where fe_person0_.name='Duck'  
select ... from FE_Telefon telefonnum0_  
    where telefonnum0_.person_fk=?  
select ... from FE_Telefon telefonnum0_  
    where telefonnum0_.person_fk=?
```

Die selben DB-Statements wie bei LAZY. Allerdings werden alle Daten sofort mit der Ausführung der Query geladen.

tst.fetch

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategie
BatchSize



Relevanter Codeausschnitt (Person)

```
private List<FE_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL)  
@JoinColumn(name = "person_fk")  
@BatchSize(size=10)  
public List<FE_Telefon> getTelefonNummern() {  
    if (telefonNummern == null) {  
        telefonNummern = new ArrayList<FE_Telefon>();  
    }  
    return telefonNummern;  
}
```

Hibernate-Log

```
4 List<FE_Telefon> telefonNummern = person.getTelefonNummern();  
--> select ... from FE_Person fe_person0_  
    where fe_person0_.name='Duck'  
  
5 for (FE_Telefon telefon : telefonNummern) {  
--> select ... from FE_Telefon telefonnum0_  
    where telefonnum0_.person_fk in (1, 2)
```

Lädt alle Telefonnummern zu den 2
Personendatensätzen in einer Abfrage.
Durch die Begrenzung der BatchSize auf 10
würde Hibernate bis maximal 10
Fremdschlüssel in dem In-Ausdruck
aufnehmen. Für weitere 10 würde erneut eine
Select-Anweisung erfolgen.

tst.fetch

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategie
SUBSELECT



Relevanter Codeausschnitt (Person)

```
private List<FE_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL)  
@JoinColumn(name = "person_fk")  
@org.hibernate.annotations.Fetch(fetchMode = FetchMode.SUBSELECT)  
public List<FE_Telefon> getTelefonNummern() {  
    if (telefonNummern == null) {  
        telefonNummern = new ArrayList<FE_Telefon>();  
    }  
  
    return telefonNummern;  
}
```

Hibernate-Log

```
4 List<FE_Telefon> telefonNummern = person.getTelefonNummern();  
--> select ... from FE_Person fe_person0  
    where fe_person0_.name='Duck'  
  
5 for (FE_Telefon telefon : telefonNummern) {  
--> select ... where telefonnum0_.person_fk  
    in (select fe_person0_.id from FE_Person fe_person0_  
        where fe_person0_.name='Duck')
```

Eine einzige Select-Anweisung für
alle Telefonnummern.

Achtung!

Meine Experimente mit `FetchMode.JOIN` und `FetchMode.SELECT` haben im Widerspruch zu deren Dokumentation keine neue Fetchstrategien gezeigt. Das könnte an der Hibernateversion oder meiner Verwendung gelegen haben.

tst.fetch

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategie JOIN FETCH

Relevanter Codeausschnitt (Person)

```
private List<FE_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL)  
@JoinColumn(name = "person_fk")  
public List<FE_Telefon> getTelefonNummern() {  
    if (telefonNummern == null) {  
        telefonNummern = new ArrayList<FE_Telefon>();  
    }  
  
    return telefonNummern;  
}
```

Genauso wie bei LAZY.

Relevanter Codeausschnitt (Query)

```
1 Query query = em.createQuery("select distinct p  
    from FE_Person p left join fetch p.telefonNummern  
    where p.name='Duck'");
```

„distinct“ wird aufgrund des
Kreuzprodukts durch den Join
notwendig.

Hibernate-Log

```
4 List<FE_Telefon> telefonNummern = person.getTelefonNummern();  
--> select distinct ... from FE_Person fe_person0_  
    left outer join FE_Telefon telefonnum1  
    on fe_person0_.id=telefonnum1.person_fk  
    where fe_person0_.name='Duck'
```

Eine einzige Select-Anweisung für
alle Personen und
Telefonnummern.

tst.fetch

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategien Vergleich

	LAZY	EAGER	BatchSize	SUBSELECT	JOIN FETCH
relevante Codestelle	Keine / default	fetch = FetchType.EAGER	@BatchSize(size=...)	@org.hibernate.annotations.Fetch(fetchMode=SUBSELECT)	select distinct ... from ... left join fetch ...
4 List<FE_Telefon> telefonNummern=person.getTelefonNummern();	select ... from FE_Person fe_person0_ where fe_person0_name='Duck'	select ... from FE_Person fe_person0_ where fe_person0_name='Duck' select ... from FE_Telefon telefonnum0_ where telefonnum0_person_fk=1 select ... from FE_Telefon telefonnum0_ where telefonnum0_person_fk=2	select ... from FE_Person fe_person0_ where fe_person0_name='Duck'	select ... from FE_Person fe_person0_ where fe_person0_name='Duck'	
5 for (FE_Telefon telefon : telefonNummern) {	select ... from FE_Telefon telefonnum0_ where telefonnum0_person_fk=1 select ... from FE_Telefon telefonnum0_ where telefonnum0_person_fk=2		select ... from FE_Telefon telefonnum0_ where telefonnum0_person_fk in (1, 2)	select ... where telefonnum0_person_fk in (select fe_person0_id from FE_Person fe_person0_ where fe_person0_name='Duck')	
ist distinct notwendig?	nein	nein	nein	nein	ja
Wann einsetzen?	Wenn man die Telefonnummern nicht oder nur selten benötigt.	Höchstens nur dann, wenn es wenig Personen mit wenig Telefonnummern gibt.	Es ist sehr empfehlenswert die BatchSize generell in der Konfiguration oder individuell an den betreffenden 1:n-Beziehungen einzustellen.		
Besonderheit	Gleiche Statements wie EAGER. Problem mit „Detached“.	Gleiche Statements wie LAZY, kein Problem mit „Detached“.		Telefonnummern werden „On Demand“ geladen. Dann aber dafür alle in einem Statement. Spezielle Eigenschaft von Hibernate.	Personen und Telefonnummern werden sofort geladen. Das Kreuzprodukt kann bei vielen Personen mit vielen Telefonnummern ungeschickter sein, als der SUBSELECT. Ausprobieren oder den DBA hinzuziehen. Im mJPA Standard enthalten.

Datenabfragen

Fetchstrategien bei 1:n Beziehungen

Fetchstrategien
Vergleich

- LAZY und EAGER führen zu naiven Datenbankabfragen.
- EAGER verhindert dabei das Problem mit „Detached“
- LAZY spielt seine Stärken aus, wenn nur selten auf Telefonnummern zugegriffen wird, oder die Anzahl der Personen klein ist.
- Generell sollte die batchSize vom Standardwert 1 auf einen höheren Wert gesetzt werden. Ggf. bei 1:n-Beziehungen individuell höher einstellen.
- Sobald optimiert werden muss, bleiben nur SUBSELECT, batchSize und JOIN FETCH.
- Hier muss man zwischen JPA Standard und Hibernate proprietär abwägen und zwischen 1 Select mit Kreuzprodukt oder 2 Selects mit Subselect oder 1 Select mit unterschiedlichevielen Folgeselects mit einer In-Clause.

Datenabfragen

ORACLE: null==,"

Null==," ?!

Achtung!

Nicht immer stehen die Daten so in der Datenbank, wie man sie ursprünglich eingefügt hatte.

Das macht sich mindestens dann unangenehm bemerkbar, wenn man auf Basis von beispielsweise MySQL entwickelt und dann für ORACLE ausliefert.

Die Kapselung durch JPA oder Hibernate ändert nichts an der unterschiedlichen Speicherung.

Fälle (HSQL)

ID	BEMERKUNG	STRING
0	Text ohne Leerzeichen.	Test
1	Text mit führendem Leerzeichen.	Test
2	Text mit anschließendem Leerzeichen.	Test
3	null anstatt Text.	<NULL>
4	Leertext "".	
5	Ein Leerzeichen.	
6	Vier Leerzeichen.	

HSQL speichert Strings wie zuvor eingefügt. Entsprechende Abfragen liefern auch das erwartete Ergebnis.

Fälle (Oracle)

ID	BEMERKUNG	STRING
0	Text ohne Leerzeichen.	Test
1	Text mit führendem Leerzeichen.	Test
2	Text mit anschließendem Leerzeichen.	Test
3	null anstatt Text.	<NULL>
4	Leertext "".	<NULL>
5	Ein Leerzeichen.	
6	Vier Leerzeichen.	

ORACLE speichert Leerstrings als NULL. Die Abfrage: `select n from NullOrEmpty as n where n.string=''`, liefert bei ORACLE keine Treffer. Die Abfrage: `select n from NullOrEmpty as n where n.string is null` liefert bei ORACLE keinen Treffer.

tst.oracle.NullOrEmpty

Arbeitsumgebung
Java-Annotations
Java Persistence API Einführung
Entity
Beziehungen
Vererbung
Datenabfragen
Sortieren
Filtern
Optimierungsmöglichkeiten
Sperrstrategien
Designempfehlungen

Sortieren

Datenbankseitiges Sortieren

@OrderBy

Person

```
private List<S_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL,  
           fetch = FetchType.EAGER)  
@JoinColumn(name = "person_fk")  
@OrderBy(value = "nummer DESC")  
public List<S_Telefon> getTelefonNummern() {  
    ...  
}
```

Telefon

```
@Entity  
public class S_Telefon {  
    ...  
    // Hier gibt es nichts besonderes  
    ...  
}
```

Testcode

```
EntityManager em = JpaUtil.getEntityManager();  
person = em.find(S_Person.class, rememberPrimaryKey);
```

Hibernate-Log

```
select s_person0_.id as id6_1_, ...  
from S_Person s_person0_ left outer join S_Telefon telefonnum1_ on s_person0_.id=telefonnum1_.person_fk  
where s_person0_.id=?  
order by telefonnum1_.nummer DESC
```

tst.sort

Sortieren

Datenbankseitiges Sortieren

Join fetch, order by

Person

```
private List<S_Telefon> telefonNummern;

@OneToMany(cascade = CascadeType.ALL,
           fetch = FetchType.EAGER)
@JoinColumn(name = "person_fk")
@OrderBy(value = "nummer DESC")
public List<S_Telefon> getTelefonNummern() {
    ...
}
```

Telefon

```
@Entity
public class S_Telefon {
    ...
    // Hier gibt es nichts besonderes
    ...
}
```

Testcode

```
EntityManager em = JpaUtil.getEntityManager();
Query q = em.createQuery("Select p from S Person p left join fetch p.telefonNummern
                        order by p.name asc");
List result = q.getResultList();
```

Hibernate-Log

```
select s_person0_.id as id6_0_, ...
from S_Person s_person0_ left outer join S_Telefon telefonnum1_ on s_person0_.id=telefonnum1_.person_fk
order by s_person0_.name asc, telefonnum1_.nummer DESC
```

tst.sort

Sortieren

Sortieren referenzierter Objekte im Arbeitsspeicher

@OrderColumn



Person

```
private List<S_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL,  
          fetch = FetchType.EAGER)  
@JoinColumn(name = "person_fk")  
@OrderColumn(name = "tel_index")  
public List<S_Telefon> getTelefonNummern() {  
    ...  
}
```

Telefon

```
@Entity  
public class S_Telefon {  
    ...  
    // Hier gibt es nichts besonderes  
    ...  
}
```

Testcode

```
EntityManager em = JpaUtil.getEntityManager();  
person = em.find(S_Person.class, rememberPrimaryKey);
```

NUMMER	BEMERKUNG	PERSON_FK	TEL_INDEX
0177-1	mobil	2	0
0177-2	mobil	2	1
0177-3	mobil	2	2
07152-2345667	privat	1	0

Hibernate-Log

```
select s_list_per0_.id as id6_0_, ...  
  from S_List_Person s_list_per0_  
  where s_list_per0_.id=?  
select telefonnum0_.person_fk as person3_6_1_, tel_index as tel4_1_, ...  
  from S_List_Telefon telefonnum0_  
  where telefonnum0_.person_fk=?
```

ID	NAME
1	Duck
2	Goofy

Indexspalte, anhand der die Reihenfolge der Telefonnummern in der Liste 'List<S_Telefon> telefonNummern' bestimmt ist.

tst.sort

Sortieren

Sortieren referenzierter Objekte im Arbeitsspeicher

@Sort



Person

```
private Set<S_Telefon> telefonNummern;  
  
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)  
@JoinColumn(name = "person_fk")  
@Sort(type=SortType.COMPARATOR, comparator=TelefonComparator.class)  
public Set<S_Telefon> getTelefonNummern() {  
    ...  
}
```

Telefon

```
@Entity  
public class S_Telefon {  
    ...  
    // Hier gibt es nichts besonderes  
    ...  
}
```

TelefonComparator

```
public class TelefonComparator implements Comparator<S_Telefon> {  
    public int compare(S_Telefon o1, S_Telefon o2) {  
        ...  
    }  
}
```

Achtung!
Das funktioniert ausschließlich mit dem
Collection-Typ Set, nicht mit List!

Testcode

```
EntityManager em = JpaUtil.getEntityManager();  
person = em.find(S_Person.class, rememberPrimaryKey);
```

Keine OrderBy-Clause, aber
dennoch eine sortierte
Telefonliste

Hibernate-Log

```
select s_person0_.id as id6_1_, ...  
from S_Person s_person0_ left outer join S_Telefon telefonnum1_ on s_person0_.id=telefonnum1_.person_fk  
where s_person0_.id=?
```

tst.sort

Arbeitsumgebung

Java-Annotations

Java Persistence API Einführung

Entity

Beziehungen

Vererbung

Datenabfragen

Sortieren

Filtern

Optimierungsmöglichkeiten

Sperrstrategien

Designempfehlungen

Filtern

Datenbankseitiges Filtern von Daten

Anwendungsfälle



- Die einzige Möglichkeit, nicht immer alle Datensätze einer Beziehung mit im Arbeitsspeicher zu haben, obwohl nur eine Gruppe von referenzierten Datensätzen von Interesse sind.
- Bilden von Sichten oder Datenpartitionen, die sich nicht durch die gesamten Datenbankabfragen ziehen sollen auf beispielsweise Bundesland, Mandant, Niederlassung, ...
- Teil einer Realisierungsstrategie für ein Soft-Delete. Bei einem Soft-Delete werden die zu löschenden Datensätze lediglich als gelöscht markiert. Mit einem Filter kann bei Datenabfragen gesteuert werden, dass die als gelöscht markierten Datensätze automatisch ausgeblendet werden.

Filtern

Datenbankseitiges Filtern von Daten

Filter an einer 1:n
Beziehung



Person

```
@Entity
@FilterDef(name = "mobil_nummern")
public class Fi_Person {
    ...
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn
    @Filter(name = "mobil_nummern",
            condition = "bemerkung = 'mobil'")
    public List<Fi_Telefon> getTelefonNummern()
    {
        ...
    }
}
```

Definiert die Existenz eines Filters mit dem symbolischen Namen „mobil_nummern“.

Legt weitere Eigenschaften für das Filter an dem 1:n Beziehungsattribut fest.
Im Beispiel wird eine einfache Bedingung (bemerkung='mobil') hinzugefügt. Sinngemäß: Liefere alle Telefonnummern, in deren Bemerkungsfeld der Wert 'mobil' drin steht.

Dieses Beispiel zeigt nur einen kleinen Ausschnitt der Möglichkeiten, die Hibernate zum Filtern bietet.

Testcode

```
EntityManager em = JpaUtil.getEntityManager();
SessionImpl session = (SessionImpl) em.getDelegate();
session.enableFilter("mobil_nummern");

Fi_Person person = em.find(Fi_Person.class, personPk);

List<Fi_Telefon>
    telefonNummern = person.getTelefonNummern();
...
```

Um ein Filter zu verwenden, muss es bei Hibernate-Session aktiviert werden.

Im Ergebnis tauchen nur Telefonnummern mit der Bemerkung 'mobil' auf. Auf die geladenen Telefonnummern können die üblichen CRUD-Operationen ausgeführt werden, ohne dass plötzlich lazy die restlichen Datensätze nachgeladen würden.

Hibernate-Log

```
Select ...
from Fi_Telefon telefonnum0
where telefonnum0.bemerkung = 'mobil' and telefonnum0_.telefonNummern_id=?
```

tst.filter

Arbeitsumgebung
Java-Annotations
Java Persistence API Einführung
Entity
Beziehungen
Vererbung
Datenabfragen
Sortieren
Filtern
Optimierungsmöglichkeiten
Sperrstrategien
Designempfehlungen

- Zum Vergleich immer die SQL-Anweisungen heranziehen, die man ohne Hibernate verwenden würde.
- Optimierungen seitens der Datenbank nicht vernachlässigen. Also Datenmodell, Indexe, partitionierte Daten, partitionierte Hash-Indexe, Was die Datenbank nicht hergibt, kann man mit Hibernate nicht wieder retten.
- Die durch Hibernate produzierten SQL-Anweisungen prüfen und kritisch hinterfragen.
- Jede Situation muss individuell betrachtet werden.
- Einige Optimierungsmöglichkeiten sind hibernatespezifisch. Für JPA 2.1 sind weitere Fetchstrategien angekündigt, die vielleicht hibernatespezifische Fetchstrategien ersetzen.
- Vermeiden von Kardinalfehlern wie beispielsweise das unnötige Laden ganzer Objektlisten, nur um die Anzahl der betreffenden Objekte festzustellen, oder exzessives Logging, oder ein Debugging JDBC-Treiber, oder ein überlasteter Connection-Pool.

- Die folgenden Stellschrauben gehen von den Standardeinstellungen LAZY-Loading für ...ToMany-Beziehungen und EAGER-Loading für ...ToOne-Beziehungen und Properties aus.
- Unbedingt die Einstellung **hibernate.default_batch_fetch_size** auf 10 oder 20 stellen. Dies reduziert bei LAZY-Loading das einzelne Nachladen referenzierter Objekte und lädt diese stattdessen in Paketen von beispielsweise 10 bzw. 20 mittels 'select ... where ... in (...)'. Zeiten bei Datenabfragen reduzieren sich teilweise durch diese Einstellung deutlich.
- LAZY vs EAGER
 - Ggf. einzelne ...ToOne-Beziehungen auf LAZY stellen
 - Sofern klar ist, dass ohnehin ein größeres Objektnetz immer vollständig geladen werden muss, können die Beziehungen auch auf EAGER-Loading eingestellt werden. Vermeidet die bei Hibernate für LAZY-Loading notwendigen Proxies.

- Seitenweiser Datenabruf/Paging; Eignet sich beispielsweise für interaktive Szenarien, bei denen ein Benutzer vor der Anwendung sitzt und durch ein Suchergebnis blättert. Alternativ kann man natürlich dem Benutzer zur Reduzierung der Ergebnismenge hinreichend lange Suchkriterien abverlangen, oder das Ergebnis einfach auf beispielsweise 100 Treffer einschränken. Der Verzicht auf die Übertragung von Datensätzen beim Paging spart:
 - Übertragungszeit nicht abgerufener Datensätze
 - Speicherplatz

- Fetchstrategie von ...ToMany-Beziehungen ausprobieren
 - Join Fetch (gemäß JPA 1.0 nicht kaskadiert verwendbar, produziert Kreuzprodukt)
 - @BatchSize (abweichend von der Standardeinstellung kann damit individuell die BatchSize beispielsweise auf 100 gesetzt werden, vermeidet Kreuzprodukt)
 - Subselect (vermeidet Kreuzprodukt)
- Filtern von ...ToMany-Beziehungen

Vermeidet das Laden von referenzierten Objekten, die für die aktuelle Datenverarbeitung nicht benötigt werden.
- Nicht alle Attribute laden
 - Projektion von Attributen
 - Ggf. LAZY Property Fetch → CGLIB
- Wahl der passenden Vererbungsstrategie
- Mit der Einstellung **hibernate.max_fetch_depth** wird die Länge der outer join Kette bei mehrstufigen ...ToOne-Beziehungen eingestellt

- Cachen von Datenobjekten zur Vermeidung von mehrfachem Laden der selben Daten.
 - Second Level Cache hält Datenobjekte anhand ihrer Primärschlüssel vor.
 - Query Cache beantwortet identische anfragen mit den selben Ergebnisdaten ohne erneute Datenabfragen. Der Second Level Cache und LAZY-Loading ist Voraussetzung für einen Query Cache.
 - Eigener Cache
 - Drängt sich für bestimmte Datenabfragen ein Cache geradezu auf, so lohnt es sich sogar gleich einen festen Cache für die Daten vorzusehen und diesen ggf. sogar vorzuladen. Eine HashMap ist immer noch schneller, als der First bzw. Second Level Cache.
- Mit der Wahl der Generatorstrategie für Primärschlüssel kann man bei der Verwendung von der Generatorstrategie TABLE im Vergleich zu einer ORACLE Sequence noch ca. 0,5ms bei jedem Insert einsparen.

- Weitere Einstellungen bei Hibernate, die sich auf die Performance auswirken könnten:
 - `hibernate.order_updates`
 - `hibernate.id.new_generator_mappings`
 - `hibernate.jdbc.fetch_size`
 - `hibernate.jdbc.use_streams_for_binary`

Optimierungsmöglichkeiten

Fallbeispiel

Ausgangssituation

- Gegenstand: SOAP, JAXB Web-Service zum Organisations- und Personendatenabruf im Zusammenhang mit einem Identitätsmanagementsystems.
- Die Messungen fanden serverseitig statt. Die Ergebnisse beinhalten auch das XML-Marshalling, um die gesamten serverseitigen Kosten darzustellen.
- Die Messwerte zur CPU-Zeit sind tatsächlich die erbrachte Arbeit und nicht die Auslastung!
- Reklamation: Eine bestimmte Abfrage nach Personen und der zugehörigen Organisationseinheit dauert zu lange.
- Abfrage in JPQL:

```
SELECT p FROM Person p
  JOIN FETCH p.identitaetHiber8 chd1
 WHERE chd1.typ = 'P' AND chd1.status = '1' AND chd1.buchkreis = '0001'
```
- Ergebnisstruktur:
n --> Person
 n --> Identität
 n --> Mail
 n --> Telekommunikation
 1 --> Reichweite
 n --> Konto
 1 --> ParentOE
 n --> Telekommunikation
 n --> Mail

Optimierungsmöglichkeiten

Fallbeispiel

Erste
Beobachtungen

- Anzahl der Treffer: 8059
- Größe des Ergebnis-XMLs: 19,9MB
- Durchlaufzeit einer Abfrage inklusive Marshalling: 57s
- ParentOE wird nicht durch einen Fremdschlüssel referenziert, sondern von der Programmlogik selbst ermittelt und nachgeladen. Hier setzen dann verschiedene Experimente mit Caches an.
- Bei 8059 Personen wird ca. 8000 mal dieselbe ParentOE in das Ergebnis unter „Person.Identity“ eingebettet.
- Es werden viele SQL-Anweisungen produziert (N+1 Problem)

Optimierungsmöglichkeiten

Fallbeispiel

Durchgeführte
Experimente

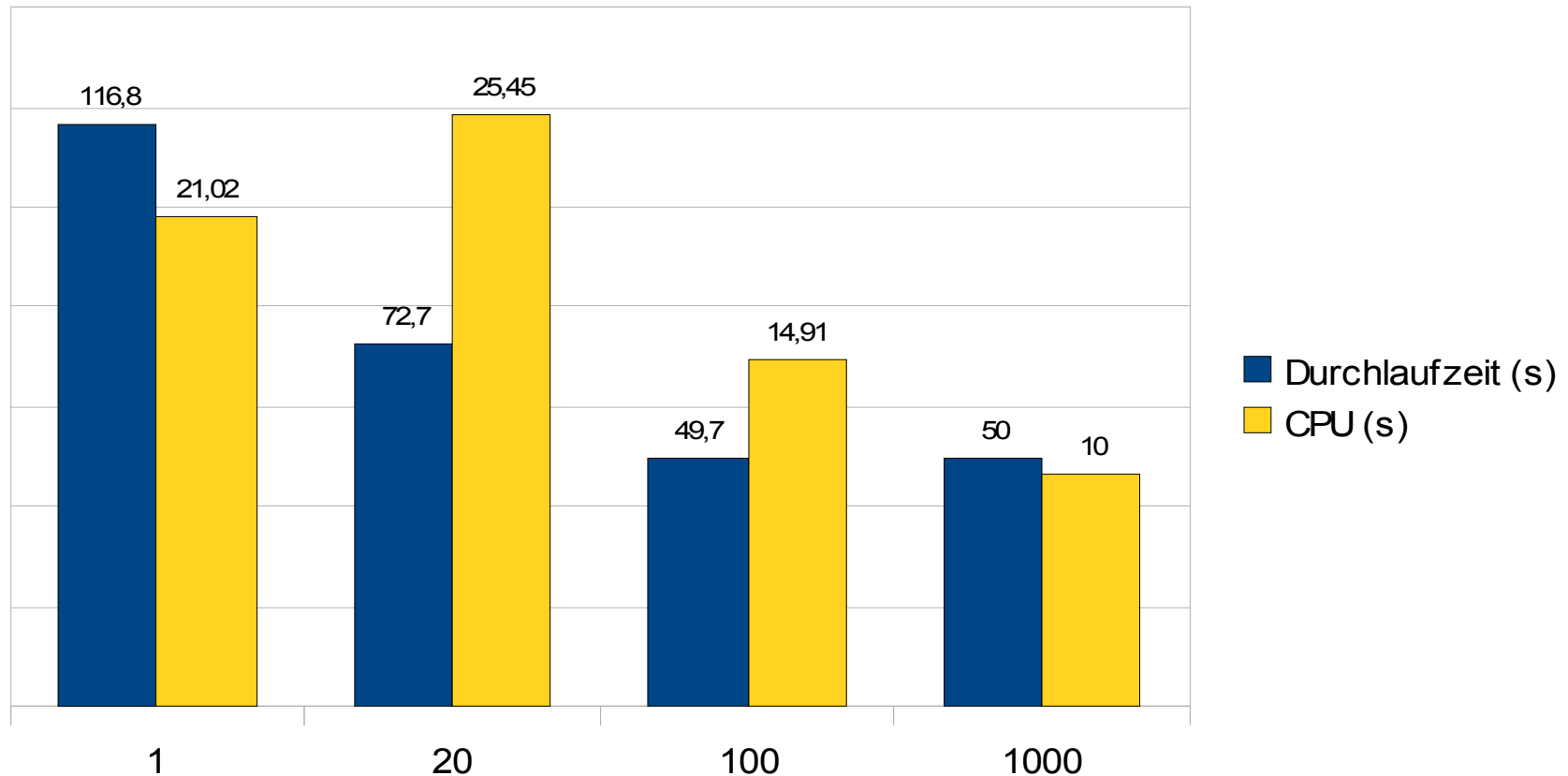
- Einstellung des Parameters „BatchSize“.
- Join Fetch bei der Beziehung Person.Identity (ja/nein).
- Seitenweiser Datenabruf/Paging (ja/nein).
- Verschiedene Varianten, OE-Objekte zu cachen:
 - Kein Cache (Abruf einer OE anhand ihres PKs via separater Query)
 - (EM) EntityManager (first level cache)
 - (HM) HashMap
 - (GC) Google Guice Cache
 - (IL) Initial Load OE

Das initiale Laden der OEn ist nicht in den Messwerten enthalten. Das Laden der OEn selbst benötigt 30s und 50MB.

Optimierungsmöglichkeiten

Fallbeispiel

Beobachtungen
BatchSize

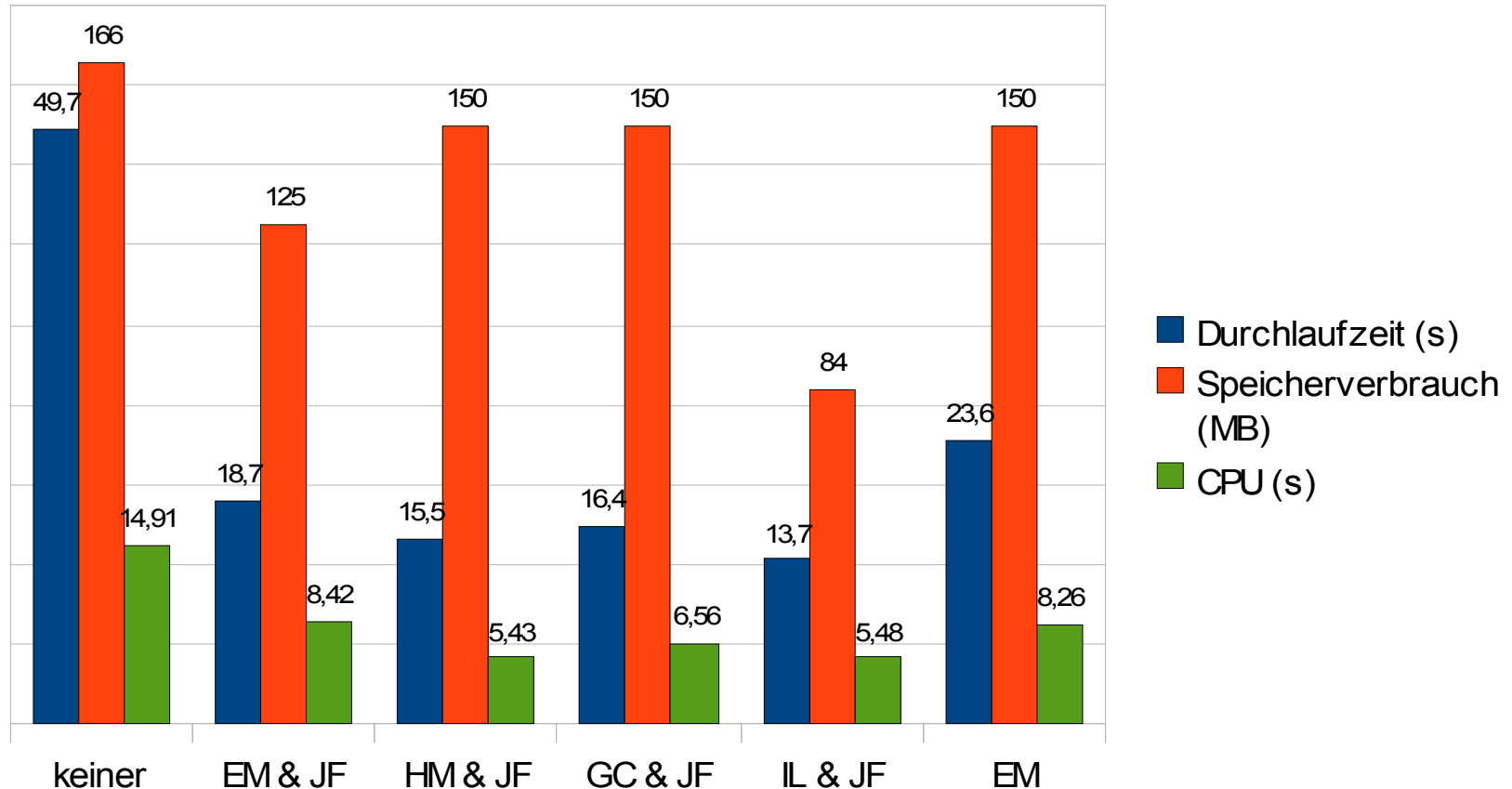


Eine BatchSize von 100 halbiert die Durchlaufzeit durch Reduzierung des (N+1) Problems.
Eine weitere Erhöhung reduziert möglicherweise (große Unschärfe) noch ein wenig die Arbeit für die CPU.

Optimierungsmöglichkeiten

Fallbeispiel

Beobachtungen
Cache f. ParentOE



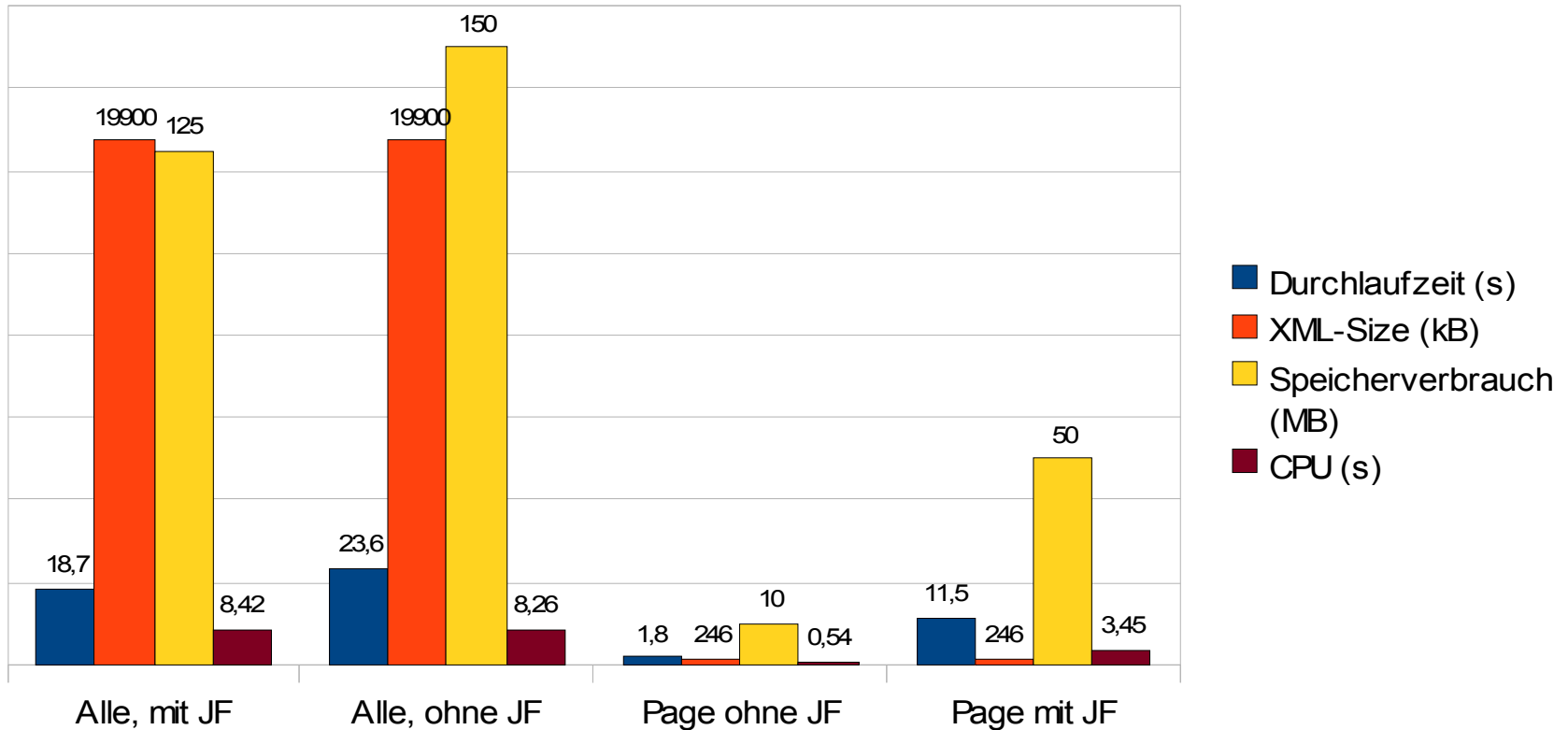
Bei „Initial Load“ sind generell zusätzlich 50MB statischer Speicher für die geladenen OEn notwendig.
Der Speicherverbrauch von „GC“ und „IL“ könnte bei mehreren Abfragen ähnlich gut aussehen.
„EM“ jedenfalls nicht!

Bei Bedarf muss der Speicherverbrauch wegen der großen Unschärfe genauer untersucht werden.
Beim Vergleich von „EM&JF“ mit „EM“ sieht man, dass (nur) hier der JF die Nase vorn hat.

Optimierungsmöglichkeiten

Fallbeispiel

Beobachtungen
Paging, Join Fetch



Der Speicherverbrauch und die XML-Größe sinken erwartungsgemäß signifikant durch das Paging. Ohne Join Fetch sind Speicherverbrauch und Durchlaufzeit beim Paging signifikant besser als mit Join Fetch.

Arbeitsumgebung
Java-Annotations
Java Persistence API Einführung
Entity
Beziehungen
Vererbung
Datenabfragen
Sortieren
Filtern
Optimierungsmöglichkeiten
Sperrstrategien
Designempfehlungen

Sperrstrategien

Transaktionale Defekte

- **Dirty-Read:** Es können von anderen Transaktionen geschriebene Daten gelesen werden, für die noch kein "Commit" erfolgte und die eventuell per "Rollback" zurückgesetzt werden.
- **Non-Repeatable-Read:** Wiederholte Lesezugriffe auf dieselben Datensätze liefern innerhalb einer Transaktion unterschiedliche Werte. Angenommen, Transaktion A liest zunächst einen Datensatz, der danach von Transaktion B geändert und durch COMMIT festgeschrieben wird. Liest Transaktion A diesen Datensatz innerhalb derselben Transaktion erneut, liefern beide Lesevorgänge unterschiedliche Werte.
- **Phantom-Read:** Bei wiederholtem Durchführen von Abfragen innerhalb einer Transaktion werden neu eingefügte und committete Datensätze anderer Transaktionen sichtbar.
- **Lost-Update (Dirty-Write):** Beim Lost-Update überschreiben sich parallel ausgeführte Transaktionen unbemerkt Daten.

Sperrstrategien

Isolationsstufen (gemäß SQL 99)

	Dirty-Read	Non-Repeatable-Read	Phantom-Read	Lost-Update
TRANSACTION_NONE				
TRANSACTION_READ_UNCOMMITTED MySQL 5.0 (InnoDB) HSQLDB 1.8.0				
TRANSACTION_READ_COMMITTED MySQL 5.0 (InnoDB) ORACLE (Standardeinstellung)	X			
TRANSACTION_REPEATABLE_READ MySQL 5.0 (Standardeinstellung bei InnoDB)	X	X		
TRANSACTION_SERIALIZABLE MySQL 5.0 (InnoDB) ORACLE	X	X	X	

X = Transaktionaler Defekt ist behoben.

Sperrstrategien

Optimistisches Verfahren

@Version

Person

```
@Version
private Long version;

public Long getVersion() {
    return version;
}
```

Zusätzliches Attribut, um Information zur Objektversion aufzunehmen.

- Für @Version können folgende Typen eingesetzt werden:
 - int, Integer
 - short, Short
 - long, Long
 - Timestamp

Testcode

```
// Transaktion A lesen
EntityManager emA = JpaUtil.getEntityManager();
L_Person personA = emA.find(L_Person.class, id);

// Transaktion B lesen
EntityManager emB = JpaUtil.getEntityManager();
L_Person personB = emB.find(L_Person.class, id);

// Transaktion B speichert
personB.setName("Transaktion B");
emB.getTransaction().begin();
emB.getTransaction().commit();

// wieder in Transaktion A
personA.setName("Transaktion A");
emA.getTransaction().begin();
emA.getTransaction().commit();
```

javax.persistence.RollbackException
caused by ...
javax.persistence.OptimisticLockException

Hibernate-Log

```
insert into L_Person (id, name, version)
values (null, 'Meier', 0)
call identity()

select l_person0_.id as id36_0_, ...,
from L_Person l_person0_
where l_person0_.id=1

select l_person0_.id as id36_0_, ...,
from L_Person l_person0_
where l_person0_.id=1

update L_Person
set name='Transaktion B', version=1
where id=1 and version=0

update L_Person
set name='Transaktion B', version=1
where id=1 and version=0
```

tst.locking

Sperrstrategien

LockMode

Durch das optimistische Sperrverfahren wird der transaktionale Defekt **Lost-Update** behandelt. Weitere transaktionale Zusicherungen können beim EntityManager durch Setzen des Lock-Modus für einzelne Datenobjekte eingefordert werden.

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

- Die JPA kennt 2 Modi:
 - **LockModeType.READ**: Er sichert zu, dass die transaktionalen Defekte Dirty-Read und Non-Repeatable-Read nicht vorkommen.
 - **LockModetype.WRITE**: Er zählt, zusätzlich zu den Zusicherungen des LockModeType.READ, den Versionszähler vorauseilend hoch.

Sperrstrategien

Pessimistisches Sperren



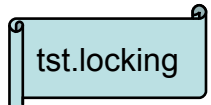
- PessimisticTest & PessimisticTransaction
 - Testklasse zum Beobachten des Sperrverhaltens zweier Transaktionen mittels Debugger.
 - Breakpoint an den Anfang von ‚Transaction.run‘ stellen und Testprogramm starten. → Beide Transaktionen bleiben im Debugger stehen.
 - Im Einzelschrittmodus des Debuggers nach belieben zwischen den Transaktionen wechseln und beobachten was passiert.

Testcode

```
EntityManager em = JpaUtil.getEntityManager();  
em.getTransaction().begin();  
  
// Beim Laden des Objektes gleich sperren  
L_Person personA = em.find(L_Person.class, id,  
                           LockModeType.PESSIMISTIC_READ);  
  
personA.setName(name);  
  
em.getTransaction().commit();
```

Hibernate-Log (ORACLE)

```
select l_person0_.id as id25_0_, ...  
  from L_Person l_person0_  
  where l_person0_.id='1' for update  
  
update L_Person set name='Transaktion B' where id='1'
```



Arbeitsumgebung
Java-Annotations
Java Persistence API Einführung
Entity
Beziehungen
Vererbung
Datenabfragen
Sortieren
Filtern
Optimierungsmöglichkeiten
Sperrstrategien
Designempfehlungen

Designempfehlungen I

- Halten Sie das Klassenmodell so einfach wie möglich.
 - Ein komplexes Modell macht auch mit einem OR-Mapper reichlich Schwierigkeiten.
 - Zur Vereinfachung des Designs sollte an Modulgrenzen auf eine modellierte Beziehung und der damit in der Datenbank verbundenen Fremdschlüssel verzichtet werden.
 - Sobald im Modell zu den hierarchischen Beziehungen noch Querbeziehungen hinzukommen, wird es in der Regel reichlich kompliziert, die Beziehungen korrekt zu pflegen. Ein Ausweg kann eine Facade sein, die die Pflege der Beziehungen komplett übernimmt.
- Erstellen Sie Richtlinien, wie Datenobjekte auszusehen haben.
 - Ansonsten können bald keine Aussagen darüber gemacht werden, wie sich die Objekte in verschiedenen Situationen verhalten.
 - Entwickler können ihr erworbenes Wissen aus anderen Modulen mit Datenobjekten nicht auf ihnen unbekannte Module übertragen.
 - Bei unterschiedlich gestalteten Datenklassen sind eigentlich Unit-Tests für alle Datenklassen und Situationen notwendig. Haben Sie Standards, so müssen nur noch die Ausnahmesituationen umfassend durch Unit-Tests abgesichert werden. Für die Standardsituationen genügen eigentlich Stichprobentests.
 - Ohne detaillierte Vorgaben und Muster wird die Persistenzanbindung unwartbar.
 - Die Verwendung von `@Temporal` genau festlegen und bei Bedarf die Richtlinie anpassen. Verlassen Sie sich insbesondere nicht auf das Defaultmapping.

Designempfehlungen II

- Designen Sie alle 1:n Beziehungen zunächst LAZY.
 - 1:1 Beziehungen werden ohnehin EAGER geladen. Das gilt übrigens auch bei Rückwärtsreferenzen, an die Sie möglicherweise gegen meine Empfehlung zuvor ein Cascade für merge oder persist annotiert haben.
 - Wenn EAGER geladen werden soll, dann verwenden Sie Join Fetch in der betreffenden Abfrage.
 - Sollte sich dennoch FetchType.EAGER für die Beziehung aufdrängen, dann dokumentieren Sie den Grund dafür!
- Erfinden Sie eine Basisklasse für alle Entitäten
 - In ihr werden alle Gemeinsamkeiten wie beispielsweise der Primärschlüssel oder auch Zeitstempelattribute zur automatischen Speicherung von Einfüge- und Aktualisierungszeitstempel festgelegt.
 - Die erfundene Basisklasse mit **@MappedSuperclass** markieren. Damit werden die Attribute der erfundenen Basisklassen bei Verwendung von **@Inheritance** nicht in einer separaten sondern in den Tabellen der betreffenden Subtypen gespeichert.

Designempfehlungen III

- Keine Cascades bei Rückwärtsreferenzen.
 - Denken Sie an das Desasterszenario aus diesem Foliensatz.
 - Vor dem Löschen (`em.remove(...)`) eines Datenobjektes immer zuerst die Beziehung korrekt ändern!
 - Sollten Sie an gut begründeten Stellen eine Ausnahme machen müssen, dann begründen Sie dies an der betreffenden Stelle im Javadoc!
- **EntityManager.flush() & .clear() nur in Sonderfällen selbst aufrufen.**
 - Kann sinnvoll eingesetzt werden, um bei einer Art Batchverarbeitung den Speicher des Persistenzkontext nicht unnötig mit Objekten anwachsen zu lassen.
 - Andere „Tricks“ mit `flush()` und `clear()` führen nur zu einem Durcheinander bei der Frage ob oder ob nicht ein Datenobjekt im Persistenzkontext ist.
 - Je nach Anwendungsfall könnte besser mit Kopien der betreffenden Datenobjekte gearbeitet werden.
- **Einsatz der JSR-303 Bean Validation**
 - Es gibt zwischen der Bean Validation und der JPA 2 überschneidende Informationen wie beispielsweise die Längenangabe von Feldern. Da die Spezifikation keine weiteren Angaben macht, müssen sicherheitshalber beide Welten und ggf. redundant mit denselben Informationen annotiert werden.