

JPA und Hibernate

Ein Open Source Persistenz-Framework für Java

JPA-2.1

OOA	OOD	OOP
OOA	OOP	OOD
OOD	OOA	OOP
OOD	OOP	OOA
OOP	OOA	OOD
OOP	OOD	OOA

Johannes Nowak
Hans-Georg Rüschenpöhler

johannes.nowak@t-online.de
ruepoe@gmx.de

August 2006
Oktober 2008
Juni 2009
April 2010
April 2011
April 2015
Oktober 2017
März 2018
September 2019

Inhalt

1	Einführung	8
1.1	Verwendete Tools	9
1.2	Installation und Quickstart	10
1.3	Projektstrukturen	15
1.4	Die x-Projekte	16
1.5	Das shared- und die util-Projekte	18
1.6	Die Kluft zwischen OO und relationalen Datenbanken	22
1.7	Aufgaben eines objekt-relationalen Mappers	23
1.8	Das Konzept von JPA	31
2	JDBC und ein einfacher OR-Mapper	32
2.1	Eine einfache JDBC-Anwendung	34
2.2	Ein einfacher OR-Mapper	37
3	JPA-Basics	42
3.1	Start	43
3.2	Mapping auf Grundlage von Attributen	56
3.3	Eine Hilfsklasse zur Konfiguration	57
3.4	Ein Transaktions-Template	59
3.5	JPA und Hibernate	65
3.6	Der Cache und die Objekt-Stat	69
3.7	Callbacks	80
3.8	Listeners	83
3.9	Eine Basisklasse BaseEntity	85
3.10	Generierte Primary-Keys	89
3.11	Generierung mittels eines TableGenerators	93
3.12	FindByBusinessKey	96
3.13	Validation	100
4	Ein kleines Service-Framework	103
4.1	Verwendung von ThreadLocal	105

4.2	DelegatingEntityManager	110
4.3	Proxies	113
4.4	Multithreading	118
5	Spezialitäten	121
5.1	Business-Keys: Composite Keys	122
5.2	Business-Keys: Embedded Composite Keys	125
5.3	FindByBusinessKey	127
5.4	Embeddable	128
5.5	Embeddable – ColumnNames	130
5.6	ElementCollection – 1	132
5.7	ElementCollection – 2	136
5.8	Secondary Tables	140
5.9	Mapping mittels XML	143
6	Queries	145
6.1	Einfache Queries	146
6.2	Parametrisierte Queries	150
6.3	Projection	152
6.4	Projection – Performance	155
6.5	Utility-Klassen: Row und RowList	156
6.6	Constructor Expressions	161
6.7	Constructor-Expressions – Performance	164
6.8	Aggregat-Funktionen	165
6.9	Bulk Update / Delete	166
6.10	Bulk Update / Delete - Performance	168
6.11	Named Queries	169
6.12	Adding Named Queries	172
6.13	Native Queries	174
6.14	Readonly Queries – Performance	176
7	Assoziationen	177
7.1	one-to-one	178
7.2	one-to-one : cascade	183

7.3	one-to-one : lazy	185
7.4	one-to-one : join	191
7.5	one-to-one : join-fetch Performance	194
7.6	one-to-one : update / delete	196
7.7	one-to-one : bidirectional	198
7.8	many-to-one	202
7.9	one-to-many	207
7.10	one-to-many, many-to-one	214
7.11	one-to-many, many-to-one : join-fetch	217
7.12	many-to-many	219
7.13	Rekursive Assoziationen	223
7.14	Alltogether	227
7.15	Projection von Entities	236
7.16	Constructor Expressions mit Entities	239
7.17	Views vs. Join-Fetch – Performance	242
7.18	Abbildung von HashMaps	244
8	Vererbung	248
8.1	Single Table	252
8.2	Joined-subclass	255
8.3	Class per Table	258
8.4	Tiefe Vererbung	261
8.5	Vererbung und Assoziationen	264
9	Versionierung und optimistische Sperren	269
9.1	Basics	270
9.2	Optimistic Locking	273
10	Stored Procedures	276
10.1	Eine einfache Procedure	277
10.2	Native Queries	278
10.3	StoredProcedureQuery	280
10.4	NamedStoredProcedureQuery	281
11	Converter – Mapping von Spalten	282

11.1	Expizites Mapping	283
11.2	Auto Apply	287
11.3	Enums	288
11.4	Calendar	290
11.5	LocalDate	292
12	Criteria	294
12.1	Einfache Queries	295
12.2	Komplexe Queries	299
12.3	Update / Delete	306
13	Entity-Graphen	308
13.1	Explizite Erzeugung	309
13.2	Named graphs	314
13.3	Mehrere Attribute	317
13.4	Subgraphs	322
13.5	Verteilte Graphen	328
13.6	Loadgraph vs. fetchgraph	331
13.7	Ein Graph-Konzept	332
14	SqlResultSetMapping	338
14.1	Constructor Expressions	339
14.2	Constructor Expressions - Associations	343
14.3	EntityResult	347
15	Metadaten	351
15.1	EntityType	352
15.2	Access	355
16	Der Second-Level Cache	357
16.1	Read-Only-Zugriffe	358
16.2	Read-Write	362
16.3	Query-Cache	364
16.4	Regions	366
17	Environments	369

17.1	Servlets	370
17.2	EJB	379
18	Übungsaufgaben	385
18.1	Start	385
18.2	ServiceDao	385
18.3	ManyToOne	386
18.4	OneToMany	386
18.5	Inheritance	387
19	Literatur	388

1 Einführung

Im folgenden werden zunächst die verwendeten Tools beschrieben.

Dann geht's um die Installation - und darum, die erste Anwendung zu übersetzen und auszuführen (Quickstart).

Anschließend wird der Aufbau des Seminarverzeichnisses und die Struktur der einzelnen Projekte beschrieben

Es werden einige Hilfsklassen vorgestellt, welche in allen Projekten durchgängig verwendet werden.

Und schließlich geht's um das Konzept des objekt-relationen Mappings.

1.1 Verwendete Tools

Die Projekte, die im folgenden vorgestellt werden, sind mit folgender Umgebung getestet worden:

JDK:

`jdk1.8.0_144` (64-Bit-VM)

Eclipse:

Oxygen

Ant:

Es wird das in Eclipse eingebundene `ant` verwendet

Hibernate / JPA:

`hibernate-release-5.2.9.Final`

Validator

`hibernate-validator-5.1.3.Final`

Logging

`log4j-1.2.15.jar`

`slf4j-log4j12-1.5.8.jar`

Datenbanken

`hsqldb-2.3.2`

`db-derby-10.14.1.0`

Tomcat

`apache-tomcat-9.0.4`

JBoss

`wildfly10`

Im Ordner `projects/dependencies` befinden sich alle erforderlichen `jar`-Dateien. Die Verweise auf diese `jar`-Dateien sind auch bereits in den Eclipse-Projekten des Seminar-Ordners eingetragen. Man braucht also keinerlei zusätzlichen Dinge, die nicht bereits im Seminar-Order enthalten wären.

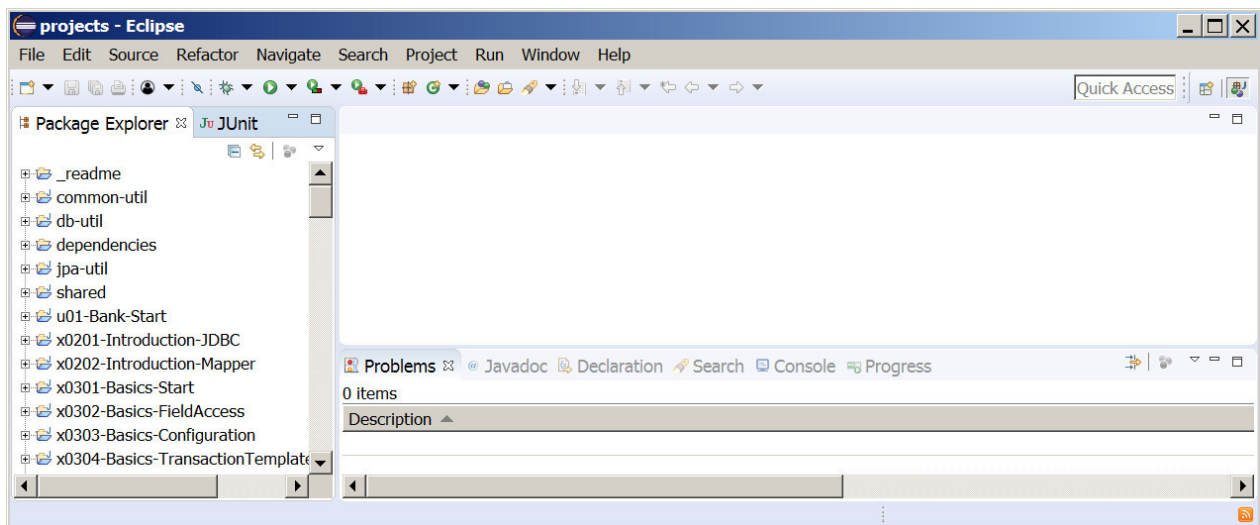
Die Beispielpprogramme des Workspaces benutzen allesamt die neuen Features von Java 8 – insbesondere die Lambda-Expressions.

1.2 Installation und Quickstart

Das Entpacken der zip-Datei kann in einem beliebigen Verzeichnis erfolgen.

Dann kann Eclipse gestartet werden – und zwar im Workspace `...\jpa-hibernate-4.3\projects`.

Im Package-Explorer von Eclipse sollten dann eine Reihe von Projekten erscheinen:

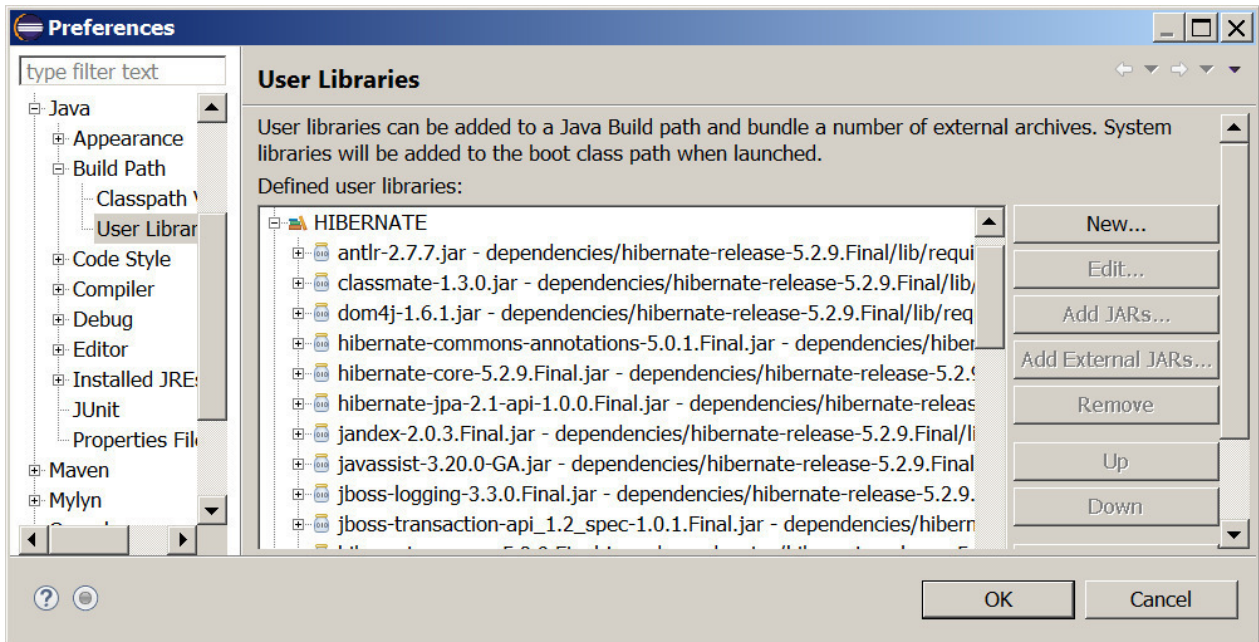


Man erkennt folgende Projekte:

```
common-util
db-util
dependencies
jpa-util
shared
x0201-Introduction-JDBC
x0202-Introduction-Mapper
...
```

Die Packages, die mit x und einer vierstelligen Nummer beginnen, seien im folgenden als x-Packages bezeichnet.

Ein Blick in Window > Preferences > Java > Build Path > User Libraries zeigt, dass in allen Projekten einige User-Libraries benutzt werden – HIBERNATE, LOGGING etc.:

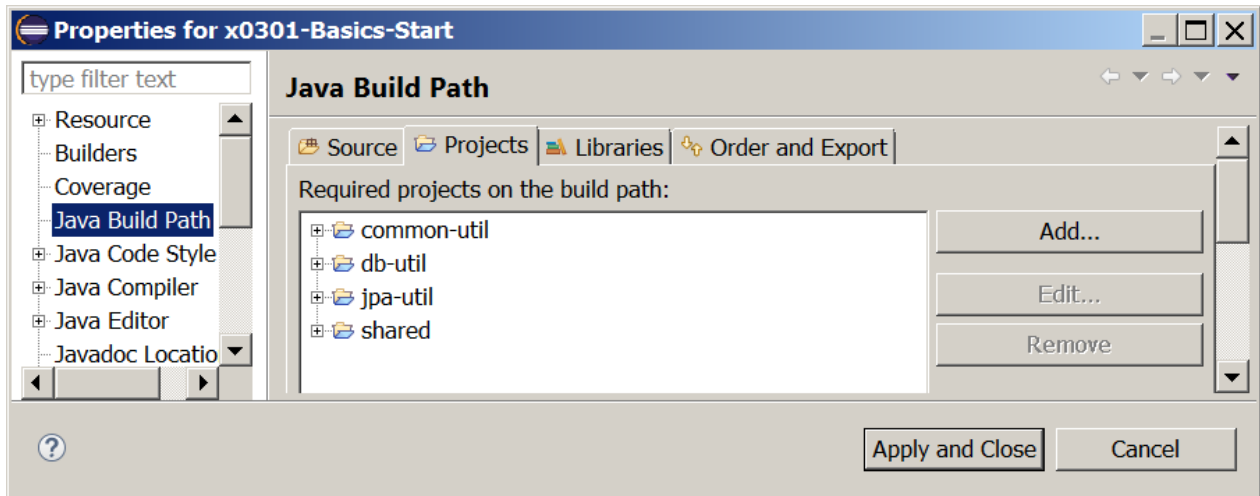


HIBERNATE enthält alle jar-Dateien von Hibernate, die für den Build und die Ausführung der Projekte erforderlich sind. Es handelt sich um die jar-Dateien aus den o.g. hibernate-release-4.3.8.Final-Ordner (unter dependencies). DATABASES enthält die jar-Dateien von HSQLDB und Derby. LOGGING enthält die jar-Dateien aus dem logging-Ordner von dependencies. TOMCAT und EJB wird für die "Environments"-Projekte genutzt werden.

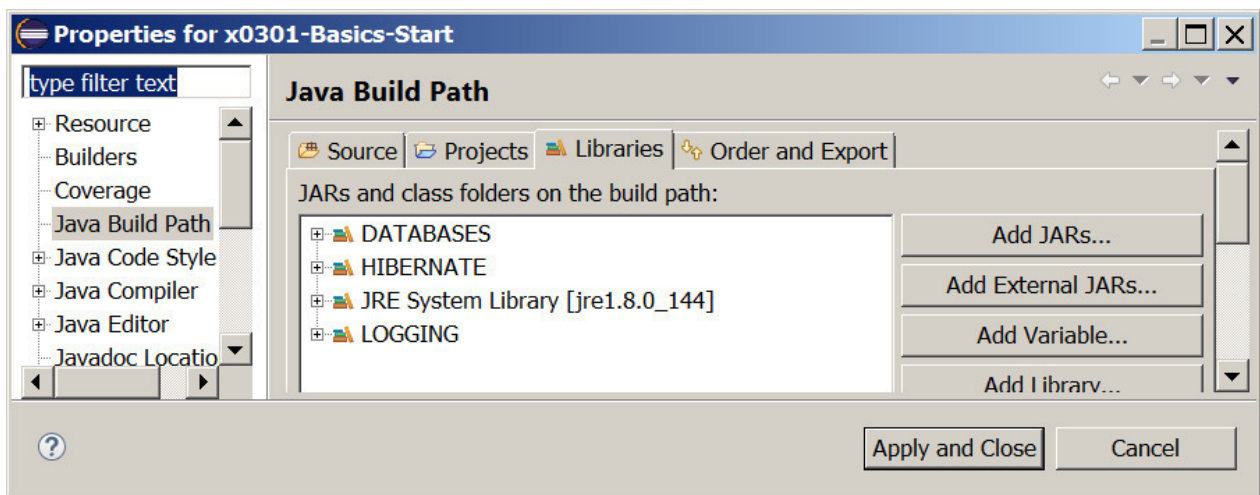
Wie man sieht, befinden sich alle benötigten externen jar-Dateien im dependencies-Ordner des Workspaces.

Jedes der x-Projekte bezieht sich auf diese User-Libraries. Und jedes dieser Projekte benutzt zusätzlich Klassen aus den Ordnern common-util, db-util und jpa-util. Diese Projekte, müssen also im CLASSPATH jedes der x-Projekte liegen. Auch das shared-Verzeichnis muss im CLASSPATH liegen.

Z.B. das erste JPA-Projekt x0301-Basics-Start: In Project > Properties > Java Build Path > Libraries finden sich Verweise auf vier Ordner:



Und es finden sich der Verweise auf die User-Libraries HIBERNATE, DATABASES und LOGGING:



Alle Beispiele des Workspaces laufen mit der Derby-Datenbank. Die meisten laufen auch mit HSQLDB. Voreingestellt ist Derby(siehe die Dateien `db.properties` und `persistence.properties` im Verzeichnis `shared/src`.)

Um nun sicher zu gehen, dass die vorbereiteten Projekte allesamt laufen, sollte das Projekt `x0301-Basics-Start` getestet werden.

Alle x-Projekte enthalten ein Package namens `appl`. Dieses enthält stets eine Klasse `Application`. Diese Klasse ist die Startklasse - über diese Klasse kann also die Anwendung gestartet werden (rechte Maustaste, Run As..., Java Application).

Dann sollte im Console-Fenster von Eclipse die folgende Ausgabe erscheinen:

```

===== db.util.batch.Executor =====
prepare [jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data]
-----

drop table BOOK
drop table PUBLISHER

all tables dropped!

CREATE TABLE BOOK (
    ISBN VARCHAR (20),
    TITLE VARCHAR (128) NOT NULL,
    PRICE DOUBLE NOT NULL,
    PRIMARY KEY (ISBN)
)
0 record(s) updated

+-----+
| demoPersist
+-----+
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
+-----+
| demoFind
+-----+
Hibernate: select ...
true
Book [2222, 20.0, Modula]
+-----+
| demoRemoveUpdate
+-----+
Hibernate: select ...
false
Hibernate: select ...
Hibernate: update Book set price=?, title=? where isbn=?
Hibernate: delete from Book where isbn=?
+-----+
| demoQuery
+-----+
Hibernate: select ...
Book [2222, 20.0, Modula-2]
Book [3333, 30.0, Oberon]

===== db.util.batch.Executor =====
select [jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data]
-----

BOOK
ISBN TITLE      PRICE
-----
2222 Modula-2  20.0
3333 Oberon    30.0

```

Damit ist sichergestellt, dass nun auch alle weiteren Anwendungen korrekt ausgeführt werden können.

1.3 Projektstrukturen

```
java-jpa-hibernate-4.3
  doc
    jpa-hibernate-4-3.pdf
  projects
    dependencies
      apache-tomcat-8.0.21
      ...
      hibernate-release-5.2.9.Final
      ...
      ...
      db-derby
      ...

    common-util
      src
      ...

    db-util
      src
      ...

    jpa-util
      src
      ...

    shared
      src
      db.properties
      persistence.properties
      ...

    x0301-Basics-Start
      src
        appl
          Application.java
        domain
          Book.java
        META-INF
          persistence.xml
          create.sql
          log4j.properties
      ...
```


1.4 Die x-Projekte

Die Namen aller x-Projekte sind wie folgt aufgebaut

- das "x"
- 2-stellige Kapitel-Nummer (z.B. 03)
- 2-stellige Abschnitts-Nummer (z.B. 01)
- Kapitel-Name (z.B. Basics)
- Abschnitts-Name (z.B. Start)

Die Kapitel- und Abschnittsnummern entsprechen der Gliederung des vorliegenden Skripts.

Alle x-Projekte sind ähnlich strukturiert:

Jedes Projekt verwaltet die Quellen in einem Ordner namens `src`.

Der Ordner `src` ist wie folgt gegliedert:

`appl` und `domain` enthalten die `java`-Quellen (und entsprechen den Java-Packages).

Der in den meisten Projekten enthaltene Ordner `META-INF` enthält die für JPA obligatorische Datei `persistence.xml` (der Name dieser Datei und deren Ort (`META-INF`) sind verpflichtend!).

Diese Datei ist die zentrale Konfigurationsdatei von JPA. Sie existiert in jedem x-Projekt und hat stets denselben Inhalt. Sie kann u.a. dazu genutzt werden, um das Logging-Verhalten von Hibernate einzustellen (`show_sql`, `format_sql`).

META-INF/persistence.xml

```
<?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="library">

    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <class>domain.Book</class>

    <properties>
```

```
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="false" />

</properties>
</persistence-unit>
</persistence>
```

`create.sql` enthält diejenigen SQL-Befehle, welche vor dem Start der jeweiligen Anwendung ausgeführt werden müssen. Sie enthält i.d.R. eine oder mehrere `CREATE TABLE`-Anweisungen. Hier z.B. die `create.sql`-Datei des ersten Projekts:

create.sql

```
create table BOOK (
    ISBN varchar (20),
    TITLE varchar (128) not null,
    PRICE double not null,
    primary key (ISBN)
);
```

1.5 Das shared- und die util-Projekte

Das shared-Projekt

Das `shared`-Projekt enthält einige Dateien, welche in allen `x`-Projekten verwendet werden.

Diese Datei `db.properties` enthält die JDBC-Verbindungsdaten (JDBC-Treiber, URL der Datenbank etc.) Sie wird von einigen Hilfs-Tools verwendet, welche vor jedem Programmlauf die Datenbanktabellen erstellen. (Hierzu im nächsten Abschnitt mehr.)

`db.properties`

```
db.driver      org.apache.derby.jdbc.EmbeddedDriver
db.url         jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data
db.user        user
db.password    password
db.schema      USER
```

`shared/src` enthält eine weitere Property-Datei: `persistence.properties`. Diese Datei enthält wiederum die JDBC-Verbindungsdaten (JDBC-Treiber, URL der Datenbank etc.) und eine "Dialekt"-Einstellung. Sie wird zur projektübergreifenden Konfiguration von JPA benutzt. (Hierzu später mehr.)

`persistence.properties`

```
javax.persistence.jdbc.driver      org.apache.derby.jdbc.EmbeddedDriver
javax.persistence.jdbc.url         jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data
javax.persistence.jdbc.user        user
javax.persistence.jdbc.password    password
hibernate.dialect                  org.hibernate.dialect.DerbyDialect
```

Das common-util-Projekt

Das Projekt `common-util` Projekt enthält u.a. eine `Util`-Klasse (im Package `common.util`), welche ausschließlich statische Methoden enthält (diese Klasse wird in fast allen `x`-Projekten genutzt):

```
package common.util;
// ...
public class Util {

    public static void mlog(Object... args) ...
```

```
public static void tlog(String text) ...  
public static String toString(Object obj, boolean verbose) ...  
public static String toString(Object obj) ...  
public static void sleep(int millis) ...  
public static void join(Thread thread) ...  
public static void join(Thread thread, long timeout) ...  
public static void wait(Object object) ...  
public static void wait(Object object, long timeout) ...  
}
```

- Mittels `mLog` kann der Einstieg einer Methode geloggt werden (dabei wird der Name der Methode, der ausgegeben wird, automatisch berechnet).
- Mittels `tLog` kann die ID des aktuellen Threads ausgegeben werden.
- `toString` kann in Methoden genutzt werden, welche die `toString`-Methode von `Object` überschreiben.
- Die weiteren Methoden rufen (statische resp. nicht-statische) `Thread`-Methoden und wickeln die dabei evtl. geworfene `InterruptedException` in eine `RuntimeException` ein.

Das Projekt enthält noch eine weitere Klasse: `Members` (im Paket `common.util`). `Members` enthält eine statische `toString(Class)`-Methode, welche die Konstruktoren, die Felder und die Attribute einer Klasse zurückliefert. Diese Klasse wird u.a. zur Untersuchung der Funktionsweise des sog. Lazy-Loadings benutzt werden.

Das db-util-Projekt

Das Verzeichnis `db-util` enthält einige Tools, die ebenfalls in jeder x-Anwendung genutzt werden:

- Ein "prepare"-Tool zur Bereinigung der Datenbank und zur Erzeugung der jeweils benötigten Tabellen. Dieses Tool wird zu Beginn jeder x-Anwendung aufgerufen. Es bereinigt die Datenbank (löscht alle vorhandenen Tabellen) und führt dann die Anweisungen der in jedem Projekt enthaltenen `create.sql`-Datei aus. Der eigentliche Programmlauf findet also immer wieder dieselben Voraussetzungen vor.

- Ein "select"-Tool zum Anzeigen aller in der Datenbank enthaltenen Tabellen. Dieses Tool wird am Ende jeder x-Anwendung aufgerufen - um sofort die Wirkungen des jeweiligen Programmlaufs studieren zu können.

Diese Tools benutzen die im `shared`-Projekt angesiedelte `db.properties`-Datei, um die JDBC-Verbindungsdaten zu ermitteln.

Am Anfang der `main`-Methode jedes der x-Projekte befindet sich folgende Zeile:

```
Db.aroundAppl();
```

Dieser Methodenaufruf führt zunächst das "prepare"-Tool aus. Dann wird ein Shutdown-Hook eingerichtet, welcher bei der Terminierung des Programms das "select"-Tool aufruft.

Hier die Klasse `db.util.appl.Db`:

Db

```
package db.util.appl;

import db.util.batch.Executor;
import db.util.logger.Logger;
import db.util.logger.NullLogger;
import db.util.logger.PrintStreamLogger;

public class Db {

    public static final String DB_PROPERTIES = "db.properties";
    public static final String CREATE_SQL = "create.sql";

    public static void aroundAppl() {
        around(DB_PROPERTIES, CREATE_SQL, new PrintStreamLogger(System.err));
    }

    public static void around(String dbPropertiesFilename,
                              String createSqlFilename, final Logger logger) {
        Executor.execute(dbPropertiesFilename, "prepare",
            createSqlFilename, logger);
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                System.out.flush();
                Executor.execute(dbPropertiesFilename, "select", null, logger);
            }
        });
    }
}
```

Das jpa-util-Projekt

`jpa-util` enthält JPA-spezifische Hilfsklassen resp. Interfaces:

```
jpa.util.Configuration  
jpa.util.DelegatingEntityManager  
jpa.util.EntityManagerThreadLocal  
jpa.util.HibernateCache  
jpa.util.QueryUtil  
jpa.util.TransactionHandler  
jpa.util.TransactionTemplate
```

Diese Klassen resp. Interfaces werden später dargestellt werden.

1.6 Die Kluft zwischen OO und relationalen Datenbanken

Die objektorientierte und die relationale Welt passen aus vielerlei Gründen nicht so recht zusammen:

- Die Objekte der Objektorientierung leben im Hauptspeicher und sind somit "flüchtig". Die "Objekte" von Datenbanken sind persistent.
- Die Objektorientierung kennt "intelligente" Objekte – Objekte, die ihren Zustand kapseln. Die Klassen dieser Objekte enthalten Methoden, mittels derer der Zustand der Objekte abfragbar und manipulierbar ist. Relationale Datenbanken dagegen enthalten nur "dumme" Daten.
- Relationale Datenbanken kennen andere Typen als objektorientierte Sprachen. Die Datenbank kennt z.B. die Typen `CHAR` und `VARCHAR`; Java dagegen kennt den Typ `String`. (Die Abbildung von SQL-Typen auf Java-Typen ist natürlich bereits in JDBC geregelt - was Standard-Typen angeht.)
- Objektorientierte Sprachen sind imperative Sprachen; die typische Datenbanksprache SQL aber ist deklarativ.
- In der objektorientierten Welt sind Objekte miteinander über Referenzen (also Pointer) verbunden. In relationalen Datenbanken werden die "Objekte" über Fremdschlüssel-Beziehungen miteinander verbunden. Die Objektorientierung kennt aber keine Fremdschlüssel (und auch keine Primärschlüssel).
- Die Objektorientierung fokussiert individuelle Objekte; zwischen diesen Objekten kann navigiert werden. (Natürlich lassen sich solche individuellen Objekte auch in Collections zusammenfassen.) Die typische Zugriffsweise von Datenbanken ist dagegen der `SELECT` in Verbindung mit dem `JOIN` – eine Zugriffsweise, die grundsätzlich Mengen von Zeilen liefert. Im Gegensatz zur Objektorientierung operiert die Datenbank also mengenorientiert.
- Die Objektorientierung kennt das Vererbungskonzept. Relationale Datenbanken dagegen kennen keine Vererbung.
- Relationale Datenbanken beruhen wesentlich auf dem Konzept der referenziellen Integrität; in der Objektorientierung ist dieses Konzept von Natur aus unbekannt.

1.7 Aufgaben eines objekt-relationalen Mappers

Aus diesen Unterschieden zwischen der objektorientierten und der Datenbank-Welt leiten sich die Hauptaufgaben eines objekt-relationalen Mappers ab.

Im folgenden wird vorausgesetzt, dass ein OR-Mapper JDBC benutzt, um auf die Datenbanken zuzugreifen.

Abbildung von Tabellenzeilen auf Objekte und umgekehrt

Ein OR-Mapper muss eine Zeile einer relationalen Tabelle auf ein Objekt abbilden können – und zwar in beide Richtungen: er muss die Spaltenwerte einer Tabellenzeile lesen und dieses dann den Attributen des Objekts zuweisen können; und er muss umgekehrt die Attributwerte eines Objekts auslesen können und diese den Spalten einer Tabellenzeile zuweisen können.

Diese Abbildung muss "generisch" erfolgen.

Auf der JDBC-Seite werden dabei die Methoden `ResultSet.getObject` und `PreparedStatement.setObject` genutzt. Mittels der `ResultSet`-Methode `getObject` kann ein beliebiger Spaltenwert aus einer Ergebnistabelle gelesen werden; `getObject` erzeugt ein zu dem Spaltentyp passendes Objekt (z.B. einen `String`, ein `Integer`- oder ein `Double`-Objekt etc.) und liefert dieses Objekt in der allgemeinen Form eines `Object`s zurück. Umgekehrt verlangt die `setObject`-Methode der Klasse `PreparedStatement` eine allgemeine `Object`-Referenz als Parameter.

```
public interface ResultSet {  
    ...  
    public Object getObject(int columnIndex)  
    public Object getObject(String columnName)  
}
```

```
public interface PreparedStatement {  
    ...  
    public void setObject(int index, Object value)  
}
```

Auf der Seite der Java-Objekte werden i.d.R. Beans (bzw. POJOs: Plain Old Java Objects) vorausgesetzt. Eine Bean ist ein Objekt einer Klasse, die erstens das Interface `Serializable` implementiert, die zweitens einen parameterlosen Konstruktor besitzt und die drittens über setter- und getter-Methoden verfügt, welche den Zugriff auf die Attribute eines Objekts dieser Klasse gestatten.

Aufgrund der Existenz eines parameterlosen Konstruktors können Objekte dann mittels `Class.newInstance` "generisch" erzeugt werden:


```
public class Class<T> {  
    ...  
    public T newInstance()  
}
```

Ein `Book`-Objekt könnte etwa wie folgt erzeugt werden:

```
String clsName = "Book";  
Class<?> cls = Class.forName(clsName);  
Object obj = cls.newInstance();
```

(Um ein Objekt einer Klasse zu erzeugen, muss also nur der Name der Klasse in Form eines `Strings` (!) bekannt sein.)

Und aufgrund der Existenz von `getter`- und `setter`-Methoden können dann die Attribute eines Objekts per `Reflection` gelesen bzw. gesetzt werden. Hier ein kleines Beispiel (wobei von der Fehlerbehandlung abgesehen wird):

```
Class<?> cls = Class.forName("Book");  
Object obj = cls.newInstance();  
....  
Method m = cls.getMethod("getTitle");  
Object value = m.invoke(obj);
```

Hier wird die Methode `setTitle` auf das zuvor erzeugte `Book`-Objekt aufgerufen. (Man beachte auch hier, dass nur der Name der Methode in Form eines `Strings` bekannt sein muss, um die Methode dann per `Method.invoke` aufrufen lassen zu können.)

Und so würde der Titel des Buches neu gesetzt werden können:

```
Method m = cls.getMethod("setTitle", String.class);  
m.invoke(obj, "Design Patterns");
```

Seien in der Klasse `Book` also z.B. folgende Methoden gegeben:

```
public class Book ... {  
    ...  
    public void setIsbn(String isbn)  
    public String getIsbn()  
  
    public void setTitle(String title)  
    public String getTitle()  
  
    public void setPrice(double price)  
    public double getPrice()  
}
```

Dann bezeichnet man die jeweilige `setter/getter`-Kombination auch als `Property`. Die Klasse `Book` enthält also die `Properties` `"isbn"`, `"title"` und `"price"`. Der Begriff

Property darf dabei nicht verwechselt werden mit dem Begriff Attribut bzw. Instanzvariable. Die Attribute, auf denen die obigen drei Methoden operieren, könnten `theke`, `antitheke` und `syntheke` heißen!

Mittels Reflection können also die Properties von Objekten gelesen und gesetzt werden.

Und schließlich muss geklärt werden, auf welche Datenbanktabelle Objekte einer Klasse abgebildet werden sollen - und welche Tabellenspalten auf welche Properties gemappt werden sollen. Die Java-Namen müssen also auf Datenbank-Namen abgebildet werden. In der Vergangenheit wurde dies häufig in zusätzlichen XML-Dateien beschrieben - etwa wie folgt:

```
<mapping class="Book" table="T_BOOK">
  <property name="isbn" column="F_ISBN" />
  <property name="name" column="F_NAME" />
  <property name="price" column="F_PRICE" />
  ...
</mapping>
```

Ein alternativer Ansatz - seit Java 5 möglich - besteht in der Verwendung von Annotations. Annotations können vom Compiler in die produzierten .class-Dateien übernommen werden, so dass sie vom OR-Mapper zur Laufzeit per Reflection ausgelesen werden können.

Die Klasse `Book` z.B. könnte durch folgende Annotations bereichert werden (das sind noch NICHT die tatsächlichen JPA-Annotations - das Beispiel dient hier nur der Veranschaulichung):

```
@Entity(table="T_BOOK")
public class Book ... {
    ...
    @Property(column="F_ISBN")
    public void setISBN (String isbn)
    public String getISBN ()

    @Property(column="F_TITLE")
    public void setTitle (String title)
    public String getTitle ()

    @Property(column="F_PRICE")
    public void setPrice (double price)
    public double getPrice ()
}
```

Abbildung von Primär-/Fremdschlüssel-Beziehungen auf Referenzen

Die in der Datenbank enthaltenen Primär-/Fremdschlüsselbeziehungen müssen auf referenzielle Beziehungen der Objekte abgebildet werden.

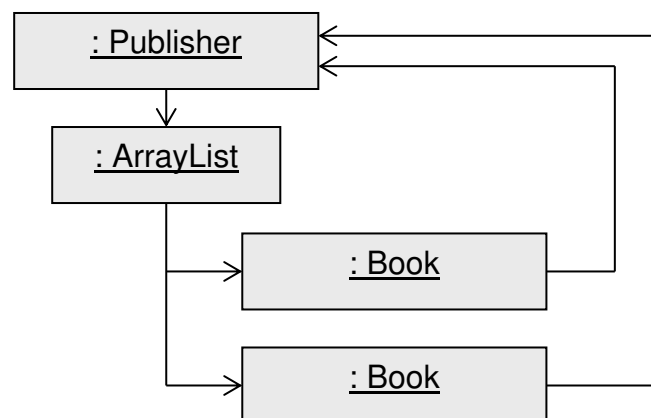
Sei z.B. neben der Tabelle `T_BOOK` noch die Tabelle `T_PUBLISHER` gegeben. Die `T_PUBLISHER`-Tabelle habe den Primärschlüssel `F_ID`. Dann würde die `T_BOOK`-Tabelle eine Fremdschlüsselspalte `F_PUBLISHER_ID` enthalten.

Hier einige beispielhafte Einträge dieser Tabellen:

T_PUBLISHER	
F_ID	F_NAME
....	
....	
5	"Addison Wesley"
....	

T_BOOK			
F_ISBN	F_TITLE	F_PRICE	F_PUBLISHER_ID
....			
....			
111111	"Design Patterns"	43.50	5
222222	"Oberon"	40.30	5
....			

Wird nun auf den Publisher mit der `F_ID` 5 zugegriffen, so sollten drei Objekte erzeugt werden: ein `Publisher`- und zwei `Book`-Objekte. Diese müssten dann etwa wie folgt miteinander verknüpft sein:



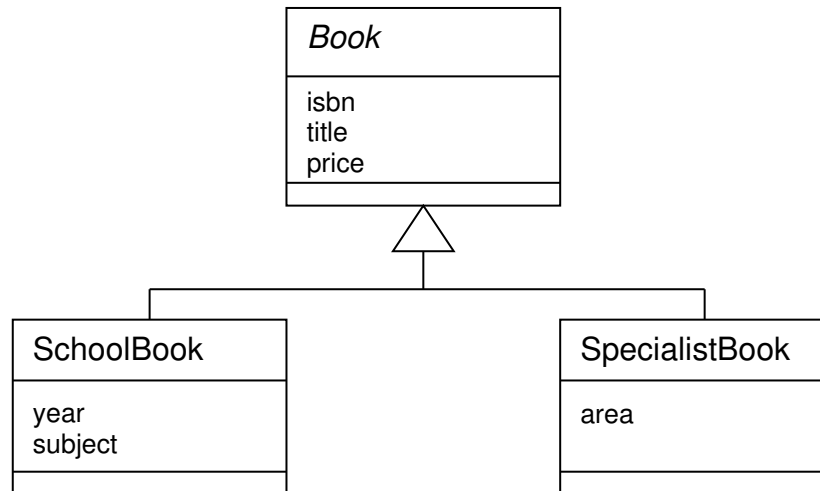
Ein `Publisher` wird ein Collection-Objekt (z.B. ein `ArrayList`) besitzen, in welchem die Referenzen auf die `Books` dieses Publishers gespeichert werden; und jedes `Book`-Objekt wird eine Referenz auf den `Publisher` besitzen, der dieses `Book` verlegt.

Der OR-Mapper sollte diese Objekte automatisch miteinander verbinden (aus Sicht der Java-Anwendung sollte die Transformation zwischen Primär-/Fremdschlüssel-

Beziehungen und den referenziellen Beziehungen der Objekte also vollständig transparent sein.)

Abbildung von Vererbungsbeziehungen

Ein OR-Mapper muss Vererbungsbeziehungen transparent auf Tabellen abbilden können. Sei z.B. folgende Klassenhierarchie gegeben:



Dann könnten Schulbücher und Fachbücher z.B. auf drei Tabellen abgebildet werden: eine Tabelle enthält die Basisdaten aller Bücher (einschließlich des Primary keys *F_ISBN*); für jede abgeleitete Klasse existiert eine weitere Tabelle, in welcher jeweils die ISBN-Nummern und die Spezialdaten der Bücher gespeichert werden.

Seien z.B. zwei Fachbücher und ein Schulbuch gegeben. Diese könnten in den Tabellen wie folgt gespeichert sein:

T_BOOK		
F_ISBN	F_TITLE	F_PRICE
111111	"Design Patterns"	43.50
222222	"Oberon"	40.30
333333	"Learning English"	5.90

T_SPECIALIST_BOOK	
F_ISBN	F_AREA
111111	"OO-Design"
222222	"OO-Programming"

T_SCHOOL_BOOK		
F_ISBN	F_YEAR	F_SUBJECT
333333	10	"English"

Um auf Grundlage dieser Tabellenstruktur ein `SpecialistBook` (oder ein `SchoolBook`) zu erzeugen, müssen dann die Daten aus zwei Tabellen eingelesen werden: aus der Basistabelle `T_BOOK` und der Tabelle `T_SPECIALISTBOOK` (bzw. `T_SCHOOLBOOK`).

Die oben gezeigte Abbildung ist nur eine von mehreren Möglichkeiten. Gleichgültig aber, welche konkrete Abbildung verwendet wird, für die Java-Anwendung sollte die Abbildung transparent sein. Die konkrete Form der Abbildung sollte auch geändert werden können, ohne dass die Java-Anwendung von einer solcher Änderung tangiert wird.

Die konkrete Abbildung dann wieder z.B. per Annotations beschrieben werden.

Generierung von SQL-Statements

Wenn dem OR-Mapping die Abbildung zwischen dem Objektmodell und dem Datenbankschema über XML-Dateien bekanntgemacht wird, dann ist es natürlich auch möglich, die für das `INSERT`, das `UPDATE` und das `DELETE` erforderlichen SQL-Statements automatisch zur Laufzeit zu generieren.

Ist z.B. das bereits oben erwähnte Annotation-basierte Mapping vorgegeben:

```
@Entity(table="T_BOOK")
public class Book ... {
    ...
    @Property(column="F_ISBN")
    public void setISBN (String isbn)
    public String getISBN ()

    @Property(column="F_TITLE")
    public void setTitle (String title)
    public String getTitle ()

    @Property(column="F_PRICE")
    public void setPrice (double price)
    public double getPrice ()
}
```

Dann kann aufgrund dieses Mappings z.B. folgender `INSERT` automatisch generiert werden:

```
INSERT INTO T_BOOK (F_ISBN, F_NAME, F_PRICE) VALUES (?, ?, ?)
```

Zum Zwecke der Generierung des `DELETE`-Statements müsste aus dem obigen Mapping allerdings auch noch hervorgehen, dass `F_ISBN` der Primary key ist. Dann könnte folgendes `DELETE`-Statement generiert werden:

```
DELETE FROM T_BOOK WHERE F_ISBN = ?
```

Objektorientierte Abfragesprache

Für Abfragen sollte ein OR-Mapper eine eigene, objektorientierte Sprache anbieten.

In der Java-Anwendung sollten die Tabellennamen und die Spaltennamen nicht bekannt sein. Die Java-Anwendung kennt nur die Properties der entsprechenden Klasse. Also sollten auch Abfragen derart formuliert werden können, dass nur die Kenntnis der Namen der Java-Klassen und die Namen der Properties dieser Klassen bekannt sein müssen.

Statt also etwa folgenden SQL-SELECT formulieren zu müssen:

```
SELECT F_ISBN, F_TITLE, F_PRICE FROM T_BOOK WHERE F_ISBN = ?
```

sollte einfach formuliert werden können:

```
select from Book b where b.isbn = ?
```

Wobei `Book` der Name der Klasse und `isbn` eine der `Book`-Properties ist. Man beachte, dass die Aufzählung der Spalten unnötig ist – denn es soll ja ein `Book`-Objekt erzeugt werden und komplett mit den Daten der entsprechenden Tabellenzeile initialisiert werden. Also muss die Projektion naturgemäß alle Spalten umfassen.

Für die Formulierung von Joins darf dann natürlich auch nicht mehr der Name der Fremdschlüsselspalte verwendet werden. Statt also zu formulieren:

```
SELECT B.ISBN, B.PRICE P.ID P.NAME  
FROM BOOK B, PUBLISHER P  
WHERE B.PUBLISHER_ID = P.PUBLISHER_ID
```

sollte etwa die folgende Zeile notiert werden können:

```
select from Book b, Publisher p where b.publisher = p
```

(Wobei vorausgesetzt wird, dass die Klasse `Book` die Methoden `getPublisher` und `setPublisher` besitzt – also die Property "publisher".)

Die objektorientierte Abfragesprache muss dann natürlich vom OR-Mapper in ein geeignetes SQL-Statement transformiert werden. Hierbei sollte beachtet werden, dass es natürlich "das" SQL überhaupt nicht gibt – vielmehr existieren unterschiedliche SQL-Dialekte. Und diese Dialekte sollten bei der Transformation berücksichtigt werden – um möglichst performante SQL-Statements zu generieren.

Identität von Objekten

Wird mittels eines OR-Mappers eine Tabellenzeile gelesen und in ein Objekt transformiert, so sollte dieses Objekt das einzige Objekt sein, welches die entsprechende Zeile im Hauptspeicher repräsentiert. Eine nochmalige Aufforderung, die entsprechende Zeile zu lesen, sollte also dasselbe Objekt zurückliefern wie die erste Aufforderung. Hierzu muss ein OR-Mapper die von ihm erzeugten und bereitgestellten Objekte geeignet cachen. Die Frage lautet dann natürlich, wie lange ein solcher Cache "gültig" bleibt.

Natives SQL

Für bestimmte Zwecke mag es sinnvoll oder gar notwendig sein, Abfragen oder DML-Befehle direkt in SQL zu formulieren. Der OR-Mapper sollte solche "workarounds" zulassen. Der OR-Mapper sollte sich also nicht einbilden, **alles** besser zu können... Insbesondere sollte natürlich der Aufruf von Stored Procedures möglich sein.

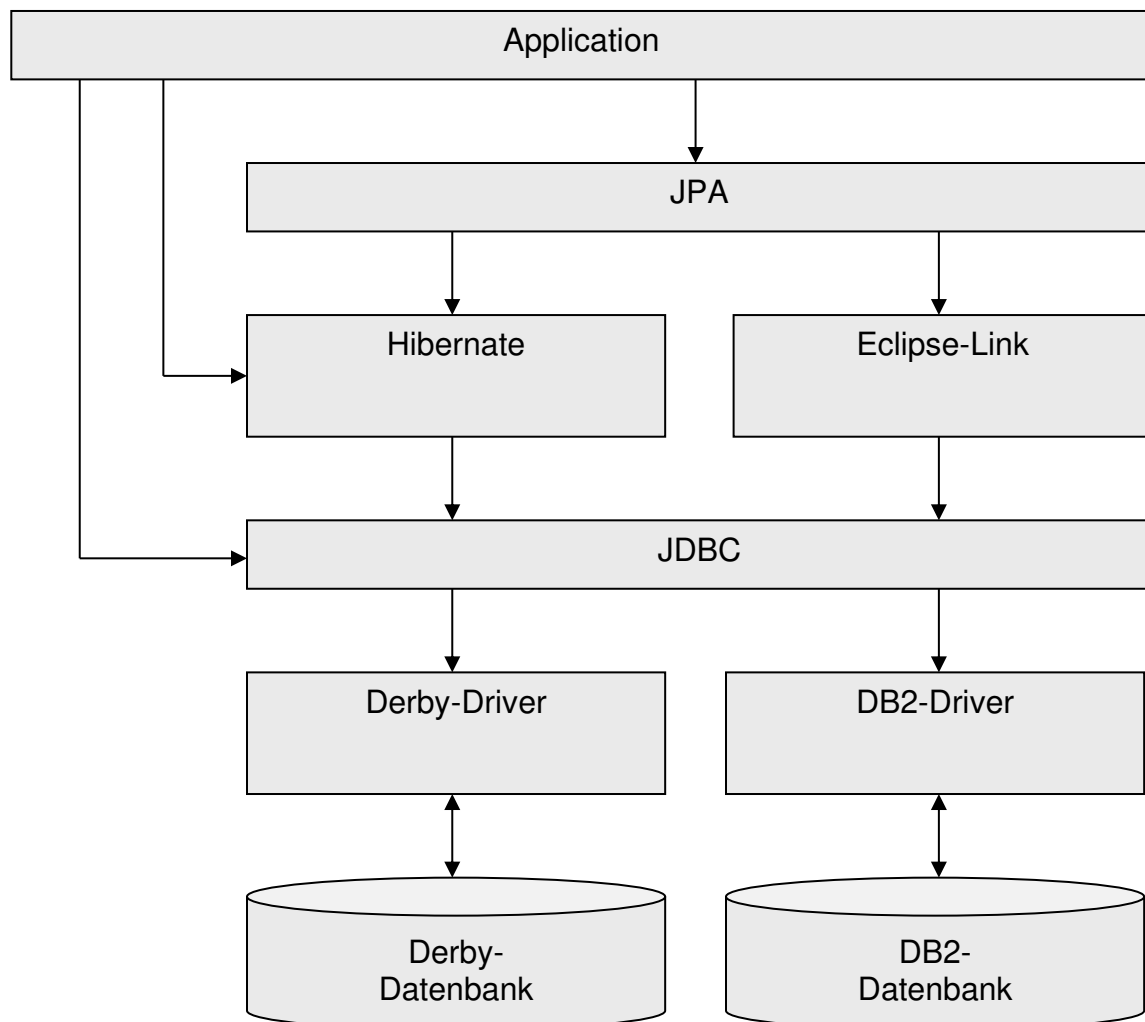
J2SE und J2EE

Ein OR-Mapper sollte sowohl in einer einfachen J2SE-Anwendung als auch in einer J2EE-Anwendung genutzt werden können. Diese Anforderung steht z.B. in direkten Widerspruch zu den sog. CMP-Entity-Beans (Container Managed Persistence) von EJB. Solche "Beans" lassen sich nur in einem EJB-Container nutzen – sind also in einer einfachen J2SE-Anwendung völlig unbrauchbar. M.a.W.: ein OR-Mapper sollte (in seinem Kern) keinerlei Annahmen über die Umgebung machen, in welcher er verwendet wird.

Keine "Verdopplung" von Datenbank-Constraints

Die Datenbank garantiert die Einhaltung bestimmter Constraints – insbesondere des `FOREIGN KEY`-Constraints (also referenzielle Integrität). Solche Constraints, die bereits die Datenbank selbst garantiert, brauchen natürlich vom OR-Mapper nicht noch zusätzlich garantiert werden. Der OR-Mapper muss also die Datenbank nicht "neu erfinden".

1.8 Das Konzept von JPA



Ähnlich wie über JDBC unterschiedliche Datenbanken weitgehend transparent angesprochen werden können, können über JPA unterschiedliche OR-Mapper angesprochen werden. Bei JDBC setzt dies voraus, dass für die jeweilige Datenbank ein entsprechender Treiber existiert (also: Implementierungen der JDBC-Interfaces!). Ein OR-Mapper, der als "JPA-Provider" fungieren kann, muss ebenfalls entsprechende Interfaces (die in JPA spezifiziert sind) implementieren.

Die Anwendung kann natürlich immer auch JPA umgehen und direkt auf den JPA-Provider zugreifen (um spezifische Eigenschaften zu nutzen, die über JPA nicht zugänglich sind). Und die Anwendung kann natürlich immer auch direkt auf JDBC-Ebene operieren...

2 JDBC und ein einfacher OR-Mapper

Im Folgenden wird zunächst eine einfache JDBC-Anwendung vorgestellt – eine Anwendung, die ein "manuelles" Mapping von `Book`-Objekten auf Zeilen einer `BOOK`-Tabelle implementiert.

Dann wird ein (sehr, sehr) einfacher OP-Mapper vorgestellt, welcher mit den Bordmitteln von Java-SE entwickelt wird – nur um zu zeigen, welche grundlegenden Techniken für die Abbildung von Objekten auf Zeilen einer relationalen Tabelle und umgekehrt erforderlich sind (und wie diese eingesetzt werden können).

Beide Anwendungen benutzen die folgende `domain`-Klasse (eine Bean-Klasse):

Book

```
package domain;

import common.util.Util;

public class Book {

    private String isbn;
    private String title;
    private double price;

    public Book() {
    }

    public Book(String isbn, String title, double price) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
    }

    public String getIsbn() { ... }
    public void setIsbn(String isbn) { ... }

    public String getTitle() { ... }
    public void setTitle(String title) { ... }

    public double getPrice() { ... }
    public void setPrice(double price) { ... }

    @Override
    public String toString() {
        return Util.toString(this);
    }
}
```

Und beide Anwendungen richten die erforderliche Datenbank-Tabelle mit folgendem CREATE-TABLE-Statements ein:

create.sql

```
CREATE TABLE BOOK (  
    ISBN VARCHAR (20),  
    TITLE VARCHAR (128) NOT NULL,  
    PRICE DOUBLE NOT NULL,  
    PRIMARY KEY (ISBN)  
);
```

Man beachte, dass die Spaltennamen den Namen der Book-Properties gleichen – ein zusätzliches Mapping von Spaltennamen auf Property-Namen ist also nicht erforderlich.

2.1 Eine einfache JDBC-Anwendung

Die folgende Anwendung fügt per `INSERT` einige Sätze zur `BOOK`-Tabelle hinzu und liest dann per `SELECT` alle Zeilen der Tabelle wieder aus:

```
package appl;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

import common.util.Util;

import db.util.appl.Db;
import domain.Book;

public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();
        final Properties props = new Properties();
        props.load(ClassLoader.getSystemResourceAsStream("db.properties"));
        Class.forName(props.getProperty("db.driver"));
        final String url = props.getProperty("db.url");
        final String user = props.getProperty("db.user");
        final String password = props.getProperty("db.password");

        demoPersist(url, user, password);
        demoQuery(url, user, password);
    }

    static void demoPersist(String url, String user, String password)
        throws Exception {
        Util.mlog();
        final String sql =
            "insert into book(isbn, title, price) values (?, ?, ?)";
        try (final Connection con =
            DriverManager.getConnection(url, user, password);
            final PreparedStatement stmt = con.prepareStatement(sql)) {
            insert(stmt, new Book("1111", "Pascal", 10));
            insert(stmt, new Book("2222", "Modula", 20));
            insert(stmt, new Book("3333", "Oberon", 30));
        }
    }

    static private void insert(PreparedStatement stmt, Book book)
        throws Exception {
        stmt.setString(1, book.getIsbn());
        stmt.setString(2, book.getTitle());
    }
}
```

```

        stmt.setDouble(3, book.getPrice());
        stmt.executeUpdate();
    }

    static void demoQuery(String url, String user, String password)
        throws Exception {
        Util.mlog();
        final String sql = "select isbn, title, price from book";
        final List<Book> books = new ArrayList<>();
        try (final Connection con =
            DriverManager.getConnection(url, user, password);
            final PreparedStatement stmt = con.prepareStatement(sql);
            final ResultSet rs = stmt.executeQuery()) {
            while(rs.next()) {
                final Book book = new Book();
                book.setIsbn(rs.getString("isbn"));
                book.setTitle(rs.getString("title"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }
        }
        books.forEach(System.out::println);
    }
}

```

Man sieht: manuelle Mappen ist "langweilig" (man beachte, dass eine "realistische" BOOK-Tabelle statt drei Spalten vielleicht 20 Spalten besitzt (und die Book-Klasse entsprechend viele Properties)...

Hier die Ausgaben des Programms:

```

===== db.util.batch.Executor =====
prepare [jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data]
-----
drop table BOOK

all tables dropped!

CREATE TABLE BOOK (
    ISBN VARCHAR (20),
    TITLE VARCHAR (128) NOT NULL,
    PRICE DOUBLE NOT NULL,
    PRIMARY KEY (ISBN)
)
0 record(s) updated

+-----+
| demoPersist
+-----+
+-----+
| demoQuery
+-----+

```

```
Book [1111, 10.0, Pascal]
Book [2222, 20.0, Modula]
Book [3333, 30.0, Oberon]
```

```
===== db.util.batch.Executor =====
select [jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data]
-----
BOOK
ISBN TITLE  PRICE
-----
1111 Pascal  10.0
2222 Modula  20.0
3333 Oberon  30.0
-----
```

2.2 Ein einfacher OR-Mapper

Das folgende Programm ist äquivalent zum letzten Programm: auch hier werden einige Zeilen zur `BOOK`-Tabelle hinzugefügt und anschließend wieder ausgelesen. Dabei wird allerdings ein kleiner, eigener Mapper verwendet.

Hier zunächst die Anwendung:

```
package appl;

import java.util.List;

import util.Mapper;
import util.MapperFactory;

import common.util.Util;

import db.util.appl.Db;
import domain.Book;

public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();
        final MapperFactory factory = new MapperFactory("db.properties");
        factory.register(Book.class);
        try {
            demoPersist(factory);
            demoQuery(factory);
        }
        finally {
            factory.close();
        }
    }

    static void demoPersist(MapperFactory factory) throws Exception {
        Util.mlog();
        final Mapper mapper = factory.createMapper();
        try {
            mapper.persist(new Book("1111", "Pascal", 10));
            mapper.persist(new Book("2222", "Modula", 20));
            mapper.persist(new Book("3333", "Oberon", 30));
        }
        finally {
            mapper.close();
        }
    }

    static void demoQuery(MapperFactory factory) throws Exception {
        Util.mlog();
        final Mapper mapper = factory.createMapper();
        try {
            final List<Book> books = mapper.getResultList(Book.class);
```

```

        books.forEach(System.out::println);
    }
    finally {
        mapper.close();
    }
}
}

```

Das Programm benutzt zwei (wiederverwendbare) Utility-Klassen.

Hier die Factory-Klasse, mittels derer der für die eigentliche Arbeit erforderliche `Mapper` erzeugt werden kann:

```

package util;

import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class MapperFactory {

    private final String url;
    private String user;
    private final String password;

    private final Map<Class<?>, PropertyDescriptor[]> registry =
        new HashMap<>();

    public MapperFactory(String propertyFilename) throws Exception {
        final Properties props = new Properties();
        props.load(ClassLoader.getResourceAsStream("db.properties"));
        Class.forName(props.getProperty("db.driver"));
        this.url = props.getProperty("db.url");
        this.user = props.getProperty("db.user");
        this.password = props.getProperty("db.password");
    }

    public void register(Class<?> cls) throws Exception {
        final PropertyDescriptor[] pds = Introspector.getBeanInfo(
            cls, Object.class).getPropertyDescriptors();
        this.registry.put(cls, pds);
    }

    public Mapper createMapper() throws Exception {
        return new Mapper(this);
    }

    public void close() {
    }

    Connection getConnection() throws Exception {
        return DriverManager.getConnection(

```

```

        this.url, this.user, this.password);
    }

    void closeConnection(Connection con) throws Exception {
        con.close();
    }

    PropertyDescriptor[] getPds(Class<?> cls) throws Exception {
        PropertyDescriptor[] pds = this.registry.get(cls);
        if (pds == null) {
            pds = Introspector.getBeanInfo(
                cls, Object.class).getPropertyDescriptors();
            this.registry.put(cls, pds);
        }
        return pds;
    }
}

```

Und hier die eigentliche Mapper-Klasse:

```

package util;

import java.beans.PropertyDescriptor;
import java.lang.reflect.Method;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Mapper {

    private final MapperFactory factory;
    private final Connection con;

    Mapper(MapperFactory factory) throws Exception {
        this.factory = factory;
        this.con = factory.getConnection();
    }

    public void persist(Object obj) throws Exception {
        final Class<?> cls = obj.getClass();
        final PropertyDescriptor[] pds = this.factory.getPds(cls);
        final String sql = buildInsertString(cls, pds);
        System.out.println("Mapper: " + sql);
        try (final PreparedStatement stmt =
            this.con.prepareStatement(sql)) {
            for(int i = 0; i < pds.length; i++) {
                final PropertyDescriptor pd = pds[i];
                final Method getter = pd.getReadMethod();
                final Object value = getter.invoke(obj);
                stmt.setObject(i + 1, value);
            }
        }
    }
}

```



```

        stmt.executeUpdate();
    }
}

public <T> List<T> getResultList(Class<T> cls) throws Exception {
    final PropertyDescriptor[] pds = this.factory.getPds(cls);
    final String sql = buildSelectString(cls, pds);
    System.out.println("Mapper: " + sql);
    final List<T> list = new ArrayList<>();
    try (final PreparedStatement stmt =
        this.con.prepareStatement(sql);
        final ResultSet rs = stmt.executeQuery()) {
        while(rs.next()) {
            final T obj = cls.newInstance();
            for(int i = 0; i < pds.length; i++) {
                final PropertyDescriptor pd = pds[i];
                final Method setter = pd.getWriteMethod();
                setter.invoke(obj, rs.getObject(i + 1));
            }
            list.add(obj);
        }
    }
    return list;
}

public void close() throws Exception {
    this.factory.closeConnection(this.con);
}

private static String buildInsertString(
    Class<?> cls, PropertyDescriptor[] pds) {
    final StringBuilder buf = new StringBuilder();
    buf.append("insert into ");
    buf.append(cls.getSimpleName());
    buf.append(" (");
    buf.append(Mapper.buildColumnList(pds));
    buf.append(") values (");
    buf.append(Mapper.buildPlaceholder(pds));
    buf.append(")");
    return buf.toString();
}

private static String buildSelectString(
    Class<?> cls, PropertyDescriptor[] pds) {
    final StringBuilder buf = new StringBuilder();
    buf.append("select ");
    buf.append(Mapper.buildColumnList(pds));
    buf.append(" from ");
    buf.append(cls.getSimpleName());
    return buf.toString();
}

private static String buildColumnList(PropertyDescriptor[] pds) {
    return Arrays.stream(pds).map(
        pd -> pd.getName()).collect(Collectors.joining(", "));
}

```

```
private static String buildPlaceholder(PropertyDescriptor[] pds) {  
    return Arrays.stream(pds).map(pd -> "?")  
        .collect(Collectors.joining(", "));  
}
```

Die beiden `java`-Dateien haben gerade einmal den Umfang von etwas 140 Zeilen – nehmen der eigentliche Anwendung aber bereits eine Menge Arbeit ab.

Die Klasse `Mapper` demonstriert insbesondere die Verwendung von Reflection und der `java.beans`-Klassen (`Introspector`, `BeanInfo`, `PropertyDescriptor`).

Das genaue Studium dieser beiden Klassen sei dem Leser / der Leserin überlassen.

3 JPA-Basics

In diesem Kapitel werden die Grundlagen von JPA vorgestellt. Hauptsächlich geht's um folgende Themen:

- Abbildung von einfachen Java-Klassen auf Tabellen
- Persistierung von Objekten und einfache Lese-Zugriffe
- Eine Template-Klasse, welche die Programmierung von Transaktionen erleichtert
- Der Zusammenhang zwischen JPA und Hibernate
- Der Hibernate-Cache
- Callbacks und Listeners
- Weitere Varianten für das Datenbank-Java-Mapping
- Validierungen

3.1 Start

Im folgenden wird gezeigt, wie mittels JPA eine einfache Java-Klasse auf eine Tabelle abgebildet wird. Diese Abbildung nutzt Annotations, die in der Java-Klasse hinterlegt sind. (Alternativ könnte auch XML für diese Abbildung genutzt werden - die Annotations-basierte Variante ist aber aus pragmatischen Gründen vorzuziehen.)

Die verwendete Tabelle wird durch folgende `CREATE TABLE` beschrieben:

create.sql

```
create table BOOK (  
    ISBN varchar (20),  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    primary key (ISBN)  
);
```

Man beachte, dass per `Db.aroundAppl` dieser `CREATE TABLE` automatisch bei jedem Start der eigentlichen Anwendung ausgeführt wird. Die Tabelle befindet sich daher nach dem Starten der Anwendung daher immer in demselben (leeren) Zustand.

Hier die Klasse `Book`, deren Objekte im folgenden persistiert werden sollen:

Book

```
package domain;  
  
import javax.persistence.Basic;  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
  
import common.util.Util;  
  
@Entity  
@Table(name = "BOOK")  
public class Book {  
  
    private String isbn;  
    private String title;  
    private double price;  
  
    Book() {  
    }  
  
    public Book(String isbn, String title, double price) {  
        this.isbn = isbn;  
        this.title = title;  
    }  
}
```

```
        this.price = price;
    }

    @Id
    @Column(name = "ISBN")
    public String getIsbn() {
        return this.isbn;
    }
    void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    @Basic
    @Column(name = "TITLE")
    public String getTitle() {
        return this.title;
    }
    public void setTitle(String title) {
        this.title = title;
    }

    @Basic
    @Column(name = "PRICE")
    public double getPrice() {
        return this.price;
    }
    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return Util.toString(this);
    }
}
```

Eine persistente Klasse (eine Klasse, deren Objekte per JPA persistiert werden können) sollte `Serializable` sein. Der Einfachheit halber wird auf die entsprechende `implements`-Klausel verzichtet werden.

Sie muss zudem zwingend einen parameterlosen Konstruktor besitzen. Dieser Konstruktor wird vom JPA-Provider (z.B. also Hibernate) verwendet, wenn dieser eine Zeile aus einer Tabelle in ein Objekt transformieren soll. Bei der Reflection-basierten Erzeugung dieses Objekts wird dann zunächst dieser parameterlose Konstruktor aufgerufen. Anschließend werden dann die setter-Methoden aufgerufen, um das Objekt zu initialisieren.

Der vom JPA-Provider benutzte parameterlose Konstruktor muss nicht `public` sein. Er kann `protected` oder `private` sein - oder die `package`-Sichtbarkeit besitzen. Natürlich ist es ratsam, die Sichtbarkeit dieses Konstruktors einzuschränken, damit er nicht von der "eigentlichen" Anwendung genutzt werden kann. Wenn die Anwendung ein `Book`

erzeugt, sollte dies natürlich über einen weiteren, parametrisierten öffentlichen Konstruktor geschehen, welcher das Objekt dann auch sofort vernünftig initialisiert.

Der JPA-Provider wird - wiederum per Reflection - die setter- resp. getter-Methoden der Klasse aufrufen, wenn er ein Objekt aufgrund der Spaltenwerte einer Tabellenzeile initialisieren soll bzw. wenn er die Daten eines Objekts zum Zwecke der Speicherung auslesen muss. Auch diese getter- und setter-Methoden müssen nicht `public` sein. Im Falle der obigen `Book`-Klasse besitzt z.B. die Methode `setIsbn` nur die `package`-Sichtbarkeit (wie auch der parameterlose Konstruktor) - damit sie von der eigentlichen Anwendung nicht aufgerufen werden kann. Sie ist somit dazu bestimmt, nur vom JPA-Provider aufgerufen zu werden.

Bekanntlich bezeichnet man ein getter-/setter-Paar auch als "Property" der Klasse (der Begriff "Property" ist also nicht mit dem Begriff "Attribut" zu verwechseln!). Die beiden Methoden `getIsbn` und `setIsbn` z.B. bezeichnen die Property "`isbn`". (Der Name einer Property leitet sich aus den Namen der getter-/setter-Namen ab - wobei jeweils `set` resp. `get` abgeschnitten wird und der erste Buchstabe in Kleinschrift konvertiert wird). Die obige `Book`-Klasse hat somit drei Properties: `isbn`, `title` und `price`. Man spricht dann auch vom Typ einer Property: der Typ einer Property ist der Return-Typ der getter-Methode resp. der Parameter-Typ der setter-Methode.

Eine persistente Klasse wird per `@Entity` angekündigt - diese Annotation ist verpflichtend. Sie kann zusätzlich eine `@Table`-Annotation besitzen, deren `name`-Attribut dann den Namen der Tabelle enthält. Sofern aber der Tabellename identisch ist mit dem (nicht qualifizierten!) Klassennamen, kann auf diese Annotation auch verzichtet werden. Im vorliegenden Falle (`BOOK == Book`) könnte also auf `@Table` verzichtet werden.

Die persistenten Properties (diejenigen Properties, die auf Spalten der Tabelle abgebildet werden sollen), werden per `@Id` resp. `@Basic` gekennzeichnet. Im obigen Falle ist die `isbn`-Property als `@Id` gekennzeichnet, die übrigen Properties als `@Basic`. Die der Primärschlüsselspalte der Tabelle entsprechende Property muss als `@Id` gekennzeichnet sein, die restlichen persistenten Properties als `@Basic`.

Die `@Id`-Annotation ist verpflichtend; `@Basic` könnte auch weggelassen werden (diese Annotation wird dann quasi automatisch als Default gesetzt).

Sowohl `@Id` als auch `@Basic` muss jeweils der getter-Methode vorangestellt werden (nicht der setter!).

Der `@Id`- und der `@Basic`-Annotation kann eine weitere `@Column`-Annotation folgen. Diese Annotation besitzt ein Attribut `name`. Es enthält den Namen der Tabellenspalte, auf welche die entsprechende Property abgebildet werden soll. Die `@Column`-Annotation kann fehlen, wenn der Name der Property und der Name der entsprechenden Spalte identisch sind (`ISBN == isbn`, `TITLE == title`, `PRICE == price`).

Im Falle der obigen `Book`-Klasse wären nur zwei Annotations zwingend erforderlich gewesen: `@Entity` und `@Id`.

(Weiterer Hinweis: enthält eine Klasse eine Property, die nicht persistent ist (zu welcher es also in der Tabelle keine entsprechende Spalte gibt), so muss diese zwingend als `@Transient` gekennzeichnet werden - denn eine Property, welche keine Annotation besitzt, gilt per default als `@Basic`!)

Konfiguration

JPA benötigt eine Konfigurationsdatei im Verzeichnis `META-INF` mit dem exakten Namen `persistence.xml`.

Man könnte im einfachsten Fall alle Konfigurationsdaten in der `persistence.xml` hinterlegen:

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

  <persistence-unit name="library">
    <provider>
      org.hibernate.ejb.HibernatePersistence
    </provider>

    <class>domain.Book</class>

    <properties>
      <property name="hibernate.connection.driver_class"
        value="..." />
      <property name="hibernate.connection.url"
        value="..." />
      <property name="hibernate.connection.username"
        value="..." />
      <property name="hibernate.connection.password"
        value="..." />
      <property name="hibernate.dialect"
        value="..." />
      <property name="hibernate.format_sql"
        value="true" />
      <property name="hibernate.show_sql"
        value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

In dieser Datei können mehrere `persistence-units` beschrieben sein. Im obigen Fall enthält die Datei genau eine solche `persistence-unit` namens "library".

Bei der Beschreibung einer `persistence-unit` können (müssen aber nicht!) die persistenten Klassen beschrieben sein (in Form von `<class>`-Elementen).

Im `<provider>`-Element wird hier der JPA-Provider eingestellt. Alle Projekte nutzen Hibernate als Provider. `org.hibernate.ejb.HibernatePersistence` ist der Name der Hibernate-"Wurzelklasse" - über diese Klasse ist Hibernate als Provider erreichbar.

Das `<properties>`-Element enthält `<property>`-Einträge, welche vom jeweiligen JPA-Provider interpretiert werden.

Hier werden insbesondere die JDBC-Verbindungsdaten beschrieben (Driver-Class, URL, User, Password).

Zusätzlich wird hier der sog. "Hibernate-Dialekt" eingestellt. Hibernate wird SQL-Statements generieren - und möchte dies so performant wie eben möglich tun. Hierbei können dann natürlich bestimmte Spezifika der verwendeten Datenbank ausgenutzt werden. Dieser Dialekteintrag ist verpflichtend.

Schließlich kann eingestellt werden, ob Hibernate die generierten SQL-Statements tracen soll oder nicht. Beim Testen ist es unbedingt empfehlenswert, den `show_sql`-Eintrag auf `true` zu setzen. (Diese `show_sql`-Ausgaben sind im Skript etwas "bereinigt" dargestellt - ohne natürlich ihren Sinn zu verfälschen!)

Einige der in der oben dargestellten `xml`-Datei eingestellten Properties sollten allerdings in einer eigenen `properties`-Datei ausgelagert werden. Es handelt sich um diejenigen Properties, welche die JDBC-Verbindungsdaten und den Hibernate-Dialekt beschreiben. Diese Daten sind projektübergreifend.

Eben zu diesen Zweck existiert die Datei `persistence.properties` (im `shared/src`-Verzeichnis):

persistence.properties:

```
javax.persistence.jdbc.driver      org.apache.derby.jdbc.EmbeddedDriver
javax.persistence.jdbc.url         jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data
javax.persistence.jdbc.user        user
javax.persistence.jdbc.password    password
hibernate.dialect                  org.hibernate.dialect.DerbyDialect
```

Diese Properties müssen dann JPA natürlich explizit bekannt gemacht werden (s. weiter unten).

Die `persistence.xml` kann dann auf den folgenden Inhalt beschränkt werden:

META-INF/persistence.xml

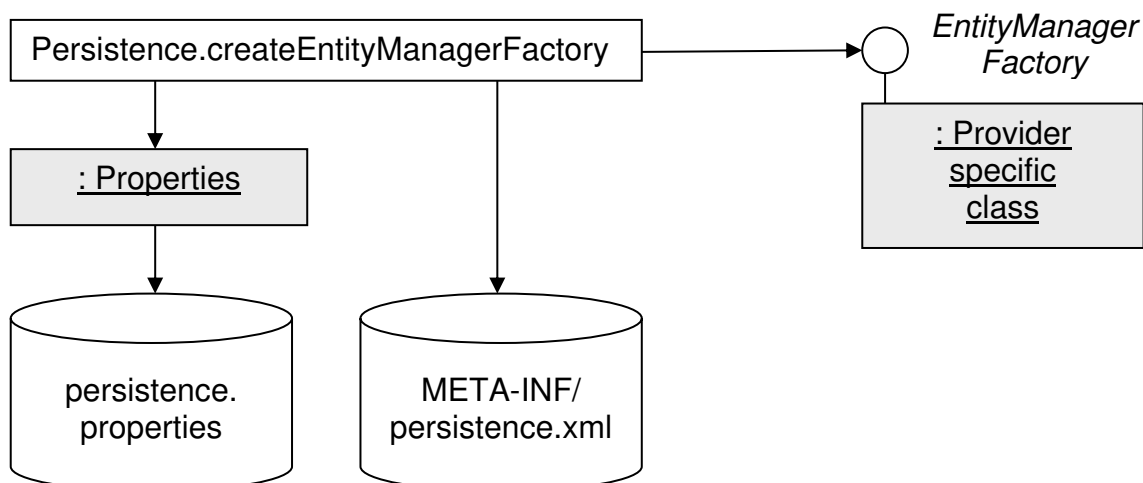
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
  <persistence-unit name="library">
    <class>domain.Book</class>
    <properties>
      <property name="hibernate.format_sql"
        value="true"/>
      <property name="hibernate.show_sql"
        value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

(Hinweis: Einträge der `persistence.xml`-Datei können durch Property-Einträge "überschrieben" werden - Properties haben also Vorrang gegenüber der `persistence.xml`).

Der Einstiegspunkt für die Nutzung von JPA ist die JPA-Klasse `Persistence`. Diese enthält eine statische Methode `createEntityManagerFactory`.

Der Methode wird einer der in der `persistence.xml` definierten `persistence-units` übergeben (hier: "library"). Der Methode kann zudem ein `Properties`-Objekt übergeben werden, welches zusätzliche Properties enthält, um welche dann die in der `xml`-Datei bereits definierten Properties erweitert werden. Und genau diese Properties werden aus der `persistence.properties`-Datei ermittelt.

Das Resultat dieses Methodenaufrufs ist eine `EntityManagerFactory`. Diese wird in der statischen Variablen `factory` gespeichert. Sie kann dann mittels des Aufrufs der statischen Methode `getEntityManagerFactory` ermittelt werden.



Hier die erste Anwendung:

Application

```
package appl;

import java.util.List;
import java.util.Properties;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

import common.util.Util;

import db.util.appl.Db;
import domain.Book;

public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        final EntityManagerFactory factory;
        final Properties props = new Properties();
        props.load(ClassLoader.getResourceAsStream(
            "persistence.properties"));
        factory = Persistence.createEntityManagerFactory(
            "library", props);

        try {
            demoPersist(factory);
            demoFind(factory);
            demoRemoveUpdate(factory);
            demoQuery(factory);
        }
        finally {
            factory.close();
        }
    }

    static void demoPersist(EntityManagerFactory factory) {
        Util.mlog();
        final EntityManager manager = factory.createEntityManager();
        final EntityTransaction transaction = manager.getTransaction();
        try {
            transaction.begin();
            manager.persist(new Book("1111", "Pascal", 10));
            manager.persist(new Book("2222", "Modula", 20));
            manager.persist(new Book("3333", "Oberon", 30));
            transaction.commit();
        }
    }
}
```

```
        catch (RuntimeException e) {
            System.out.println(e);
            if (transaction.isActive())
                transaction.rollback();
            throw e;
        }
        finally {
            manager.close();
        }
    }

    static void demoFind(EntityManagerFactory factory) {
        Util.mlog();
        final EntityManager manager = factory.createEntityManager();
        final EntityTransaction transaction = manager.getTransaction();
        try {
            transaction.begin();
            final Book book = manager.find(Book.class, "2222");
            final Book book1 = manager.find(Book.class, "2222");
            System.out.println(book == book1);
            System.out.println(book);
            transaction.commit();
        }
        catch (RuntimeException e) {
            System.out.println(e);
            if (transaction.isActive())
                transaction.rollback();
            throw e;
        }
        finally {
            manager.close();
        }
    }

    static void demoRemoveUpdate(EntityManagerFactory factory) {
        Util.mlog();
        final EntityManager manager = factory.createEntityManager();
        final EntityTransaction transaction = manager.getTransaction();
        try {
            transaction.begin();
            final Book book1 = manager.find(Book.class, "1111");
            manager.remove(book1);
            final Book book2 = manager.find(Book.class, "2222");
            book2.setTitle("Modula-2");
            transaction.commit();
        }
        catch (RuntimeException e) {
            System.out.println(e);
            if (transaction.isActive())
                transaction.rollback();
            throw e;
        }
        finally {
            manager.close();
        }
    }
}
```

```

static void demoQuery(EntityManagerFactory factory) {
    Util.mlog();
    final EntityManager manager = factory.createEntityManager();
    final EntityTransaction transaction = manager.getTransaction();
    final List<Book> bookList;
    try {
        transaction.begin();
        final TypedQuery<Book> query =
            manager.createQuery("select b from Book b", Book.class);
        bookList = query.getResultList();
        transaction.commit();
    }
    catch (RuntimeException e) {
        System.out.println(e);
        if (transaction.isActive())
            transaction.rollback();
        throw e;
    }
    finally {
        manager.close();
    }
    bookList.forEach(System.out::println);
}
}

```

Zunächst wird dafür gesorgt, dass zu Beginn die Datenbank bereinigt wird und die für die Anwendung erforderlichen Tabellen angelegt werden:

```
Db.aroundAppl();
```

Wie oben bereits beschrieben, richtet dieser Methodenaufruf zusätzlich einen ShutdownHook ein, welcher am Ende des Programms alle Tabellen der Datenbank ausgeben wird.

Dann wird die `EntityManagerFactory` erzeugt (in einer Anwendung sollte es genau eine einzige, "global" nutzbare Instanz dieser Klasse geben):

```

final EntityManagerFactory factory;
final Properties props = new Properties();
props.load(ClassLoader.getResourceAsStream(
    "persistence.properties"));
factory = Persistence.createEntityManagerFactory(
    "library", props);

```

Die `EntityManagerFactory` wird anschließend an vier Demo-Methoden übergeben: `demoPersist`, `demoFind`, `demoRemoveUpdate` und `demoQuery`.

In jeder der Demo-Methoden wird mittels der `EntityManagerFactory` zunächst ein `EntityManager` erzeugt - mittels der Methode `createEntityManager`. Diese Methode erzeugt ein Objekt einer Klasse, die das Interface `EntityManager` implementiert. Und

mit diesem `EntityManager` könnte man nun die eigentlichen Aufgaben erledigen. Am Ende von `main` wird die `EntityManagerFactory` mit `close` geschlossen.

Jede der Demo-Methoden erzeugt neben einem `EntityManager` eine Transaktion. (Für jede Transaktion sollte ein eigener `EntityManager` zuständig sein. Es wäre zwar prinzipiell auch möglich, mittels ein und desselben `EntityManager`s mehrere Transaktionen auszuführen – aus Gründen, die später erläutert werden, sollte mit einem `EntityManager` aber nur genau eine Transaktion ausgeführt werden (die Lebenszeiten von `EntityManager`n und Transaktionen sollten also aneinander gekoppelt sein)).

Alle die in der obigen Anwendung implementierten Transaktionen haben jeweils dieselbe grundlegende Form:

```
final EntityManager manager = factory.createEntityManager();
final EntityTransaction transaction = manager.getTransaction();
try {
    transaction.begin();

    // ...

    transaction.commit();
}
catch (RuntimeException e) {
    System.out.println(e);
    if (transaction.isActive())
        transaction.rollback();
    throw e;
}
finally {
    manager.close();
}
```

Zunächst wird jeweils ein `EntityManager` erzeugt. Mittels dieses `EntityManager`s wird ein Transaktions-Objekt erzeugt (ein Objekt, dessen Klasse das Interface `EntityTransaction` implementiert). Mittels `begin` wird dann die Transaktion gestartet. Auch Transaktionen, die nur Lesezugriffe ausführen, sollten im Kontext einer Transaktion programmiert werden.

Hinweis: Das `EntityTransaction`-Objekt repräsentiert hier eine einfache JDBC-Transaktion - in einem Application-Server wäre dies eine JTA-Transaktion.

Dann kann man mittels des `EntityManager`s die eigentlichen Aufgaben ausführen.

Am Ende wird die Transaktion mittels `commit` festgeschrieben – bzw. im `catch`-Zweig mittels `rollback` zurückgesetzt (sofern sie nicht bereits zuvor von Hibernate zurückgesetzt wurde – sofern als `isActive` den Wert `true` liefert).

Schließlich wird im `finally`-Block jeweils dafür gesorgt, dass der `EntityManager` mittels `close` geschlossen wird.

Dieser hier vorgestellte Rahmen ist bei allen Transaktionen derselbe. Und es ist natürlich langweilig (und fehleranfällig!), diesen Rahmen immer wieder neu zu programmieren (copy & paste!). Und man kann wegen des ganzen "Drumherums" den eigentlichen "Kern" der Transaktionen kaum erkennen... Man wird also darüber nachdenken müssen, wie dieser Rahmen geeignet abstrahiert werden kann – wie man also ein kleines "Mini-Framework" schreiben kann, in welchem dieser Rahmen nur ein einziges Mal implementiert wird. Siehe hierzu das nächste Kapitel.

Zunächst aber zu den Transaktionen der obigen Beispielanwendung.

Die erste Transaktion persistiert einige `Book`-Objekte:

```
manager.persist(new Book("1111", "Pascal", 10));
manager.persist(new Book("2222", "Modula", 20));
manager.persist(new Book("3333", "Oberon", 30));
```

Das erzeugte `Book` wird jeweils an die `persist`-Methode des `EntityManagers` übergeben. Diese kümmert sich um alles weitere (insbesondere natürlich um das Absetzen der erforderlichen SQL-INSERT-Statements)...

Die zweite Transaktion (in der Demo-Methode `demoFind`) führt einen Lesezugriff aus (Direktzugriff mittels des Primary Keys):

```
final Book book = manager.find(Book.class, "2222");
```

An die `EntityManager`-Methode `find` wird die Klasse des von Hibernate zu erzeugenden Objekts (hier: `Book.class`) und der Primary Key übergeben. Hibernate wird einen `SELECT` ausführen, das Ergebnis dieses `SELECTS` (eine Tabellenzeile) in ein `Book` transformieren und dieses `Book` zurückliefern. Das zurückgelieferte `Book` wird nach Beendigung der Transaktion ausgegeben (siehe den kompletten Programmcode).

Wird die `find`-Methode das zweite Mal aufgerufen:

```
final Book book1 = manager.find(Book.class, "2222");
System.out.println(book == book1);
```

dann wird dasselbe(!) `Book` zurückgeliefert wie bei ersten Aufruf. Der Referenzvergleich liefert also `true`.

Die Transaktion in `demoRemoveUpdate` löscht das "Pascal"-Buch und ändert den Title des "Modula"-Buches:

```
final Book book1 = manager.find(Book.class, "1111");
manager.remove(book1);
```

```
final Book book2 = manager.find(Book.class, "2222");
book2.setTitle("Modula-2");
```

Per `find` wird jeweils zunächst das jeweilige `Book` ermittelt. Das erste `Book` wird an `remove` übergeben – Hibernate wird einen SQL-DELETE absetzen. Auf das zweite `Book` wird nur die Methode `setTitle` aufgerufen – diese Änderung hat aber "durchschlagende" Wirkung: am Ende der Transaktion wird Hibernate einen SQL-UPDATE absetzen. Wie dies alles funktioniert, wird später ausführlich erläutert werden...

In der Transaktion von `demoQuery` wird schließlich die Liste aller Bücher ermittelt:

```
final TypedQuery<Book> query =
    manager.createQuery("select b from Book b", Book.class);
bookList = query.getResultList();
```

Dabei ist `bookList` vom Typ `List<Book>`. Diese Liste wird nach Beendigung der Transaktion ausgegeben:

```
bookList.forEach(System.out::println);
```

An die `createQuery`-Methode des `EntityManagers` wird ein Abfrage-Statement übergeben und die Klasse der zu erzeugenden Objekte (`Book.class`). Der Abfrage-String ist kein(!) SQL-String – es handelt sich vielmehr um die JPA-Query-Language (eine "objektorientierte" Abfragesprache, die bezüglich der Syntax aber große Ähnlichkeiten mit SQL aufweist). Alle Namen, die in einer solchen Anweisung verwendet werden, sind Java-Namen – und keine Datenbankbegriffe. In der obigen Anfrage meint `Book` also die Java-Klasse – und nicht den Namen der Tabelle!

`createQuery` liefert ein `TypedQuery`-Objekt zurück (ein Objekt, dessen Klasse das Interface `TypedQuery` implementiert). Auf dieses wird `getResultList` aufgerufen – wobei diese Methode dann den `SELECT` absetzt und dessen Resultat in eine `List<Book>` transformiert und diese `List` zurückliefert.

Der Query-Mechanismus wird später natürlich wesentlich ausführlicher diskutiert werden.

Das obige Programm erzeugt folgende Ausgaben (sie sind – der besseren Lesbarkeit halber – etwas "bereinigt"):

```
===== db.util.batch.Executor =====
prepare [jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data]
-----
drop table BOOK

all tables dropped!

CREATE TABLE BOOK (
    ISBN VARCHAR (20),
```

```

        TITLE VARCHAR (128) NOT NULL,
        PRICE DOUBLE NOT NULL,
        PRIMARY KEY (ISBN)
    )
0 record(s) updated

+-----+
| demoPersist
+-----+
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
+-----+
| demoFind
+-----+
Hibernate: select b.isbn, b.price, b.title from Book b where b.isbn=?
true
Book [2222, 20.0, Modula]
+-----+
| demoRemoveUpdate
+-----+
Hibernate: select b.isbn, b.price, b.title from Book b where b.isbn=?
Hibernate: select b.isbn, b.price, b.title from Book b where b.isbn=?
Hibernate: update Book set price=?, title=? where isbn=?
Hibernate: delete from Book where isbn=?
+-----+
| demoQuery
+-----+
Hibernate: select b.isbn, b.price, b.title from Book b
Book [2222, 20.0, Modula-2]
Book [3333, 30.0, Oberon]

===== db.util.batch.Executor =====
select [jdbc:derby:../dependencies/db-derby-10.14.1.0-lib/data]
-----
BOOK
ISBN TITLE      PRICE
-----
2222 Modula-2  20.0
3333 Oberon    30.0
-----

```

Die Zeilen, die mit `Hibernate:` beginnen, sind "bereinigte" Trace-Ausgaben von Hibernate ("`show_sql`" ist auf `true` gesetzt).

Die Ausgaben der "prepare"- und "select"-Tools werden über `System.err` produziert (erscheinen in Eclipse daher rot).

3.2 Mapping auf Grundlage von Attributen

Hibernate kann als JPA-Provider auch direkt auf die Felder eines Objekts zugreifen - statt über den "Umweg" der Properties der entsprechenden Klasse zu gehen. Diese Felder können auch `private` sein.

Book

```
package domain;
// ...
@Entity
public class Book {

    @Id
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    Book() {
    }

    public Book(String isbn, String title, double price) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
    }

    // getter / setter / toString
}
```

Natürlich könnte `@Basic` auch hier wieder fehlen...

Die Hibernate-Dokumentation empfiehlt, die getter-Methoden der Properties zu annotieren. Aus Gründen der Übersichtlichkeit werden in allen folgenden Beispielen aber die Attribute annotiert werden (zumal dies auch die gängige Praxis ist).

3.3 Eine Hilfsklasse zur Konfiguration

Für jede JPA-Anwendung benötigt man eine `EntityManagerFactory` – und zwar exakt eine einzige Instanz dieser Klasse. Die Anwendung muss überall auf diese Instanz zugreifen können, um `EntityManager` erzeugen zu können.

Hier bietet sich die Verwendung einer Hilfsklasse an (implementiert im Projekt `jpa-util`):

JPAConfig

```
package jpa.util;

import java.util.Properties;

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Configuration {

    private static final EntityManagerFactory factory;

    static {
        try {
            final Properties props = new Properties();
            props.load(ClassLoader.getResourceAsStream(
                "persistence.properties"));
            factory = Persistence.createEntityManagerFactory(
                "library", props);
        }
        catch(final Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static EntityManagerFactory getEntityManagerFactory() {
        return factory;
    }
}
```

Beim Laden der Klasse (im statischen Block) wird die `EntityManagerFactory` erzeugt. Sie wird an die statische Variable `factory` gebunden und ist über den Aufruf der statischen Methode `getEntityManagerFactory` global erreichbar.

Man beachte, dass diese Lösung nur in einer Standalone-Anwendung funktioniert. Bei einer Web-Anwendung benötigt man eine andere Lösung. Dort könnte z.B. eine `ContextListener` dazu verwendet werden, die Factory zu erzeugen und als Attribut in den `ServletContext` einzutragen. Derselbe `ContextListener` könnte die Factory

dann auch wieder schließen. Und in einer EJB-Anwendung ist diese Factory via JNDI erreichbar. Siehe hierzu die Projekte des Kapitels "Environments".

Hier die etwas verkürzte Anwendung:

Application

```
package appl;
// ...
import jpa.util.Configuration;

public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();

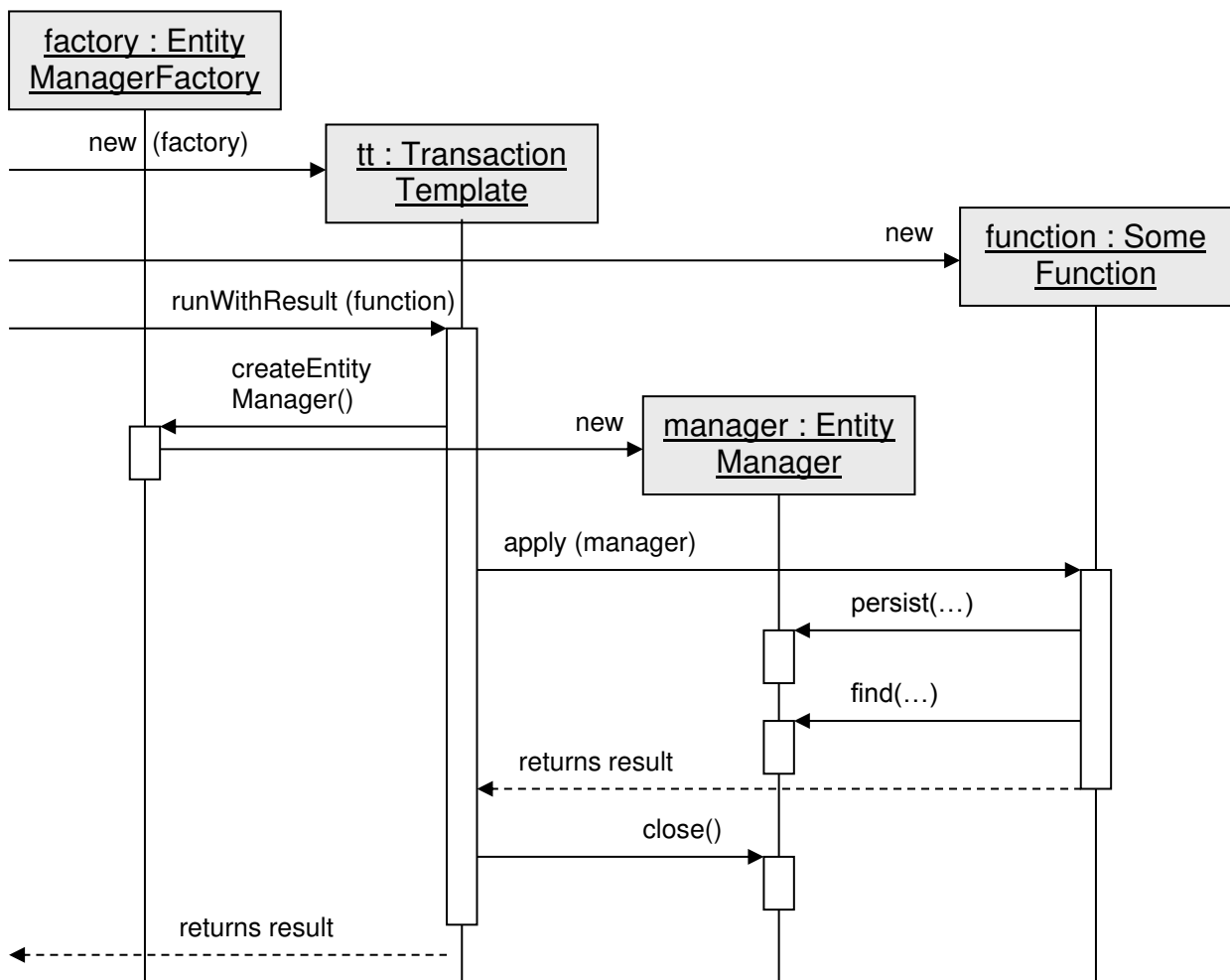
        final EntityManagerFactory factory =
            Configuration.getEntityManagerFactory();
        try {
            demoPersist(factory);
            demoFind(factory);
            demoRemoveUpdate(factory);
            demoQuery(factory);
        }
        finally {
            factory.close();
        }
    }

    // ...
}
```

3.4 Ein Transaktions-Template

Im folgenden wird gezeigt, wie die prinzipielle Form einer Transaktion (s.o.) abstrahiert werden kann. Es wird eine Klasse `TransactionTemplate` entwickelt, welche diese allgemeine Form implementiert – in Form einer `runWithResult`-Methode, welche an ein Objekt einer anwendungsspezifischen Klasse delegiert, die ein `Function`-Interface implementiert. Diese Klasse muss dann nurmehr den eigentlichen Inhalt der Transaktion implementieren. (Die Klasse befindet sich im Projekt `jpa-util`.)

Hier zunächst ein Sequenzdiagramm:



Die Klasse `TransactionTemplate` definiert zunächst ein eigenes `Function`-Interfaces:

```
public class TransactionTemplate {
    public interface Function<T>
```

```

        extends java.util.function.Function<EntityManager, T> {
    }
    // ...
}

```

Zunächst wird ein `TransactionTemplate` erzeugt. Dem Konstruktor wird der Verweis auf die `EntityManagerFactory` mitgegeben.

Dann wird ein Objekt einer Klasse (hier: `SomeFunction`) erzeugt, welche das Interface `TransactionTemplate.Function<T>` implementiert. Dieses Interface ist wie folgt definiert:

```

public class TransactionTemplate {

    @FunctionalInterface
    public interface Function<T> {
        public abstract T apply(EntityManager manager);
    }
    // ...
}

```

Dann wird schließlich die `runWithResult`-Methode auf das `TransactionTemplate` aufgerufen, welcher dieses `Function`-Objekt als Parameter übergeben wird.

Diese `runWithResult`-Methode abstrahiert die allgemeine Form des Ablaufs einer Transaktion. Zunächst wird unter Zuhilfenahme der `EntityManagerFactory` ein `EntityManager` erzeugt. Mittels dieses `EntityManagers` wird eine Transaktion erzeugt (und diese wird gestartet). Nach diesen vorbereitenden Aktionen wird die `apply`-Methode von `SomeFunction` aufgerufen, welcher der zuvor erzeugte `EntityManager` als Parameter übergeben wird.

Die `apply`-Methode von `SomeFunction` kann dann unter Zuhilfenahme des ihr übergebenen `EntityManagers` die anwendungsspezifischen Aufgaben ausführen (persist, find, ...) Sie liefert ein Ergebnis zurück (z.B. das Ergebnis eines find-Aufrufs).

Die Kontrolle kehrt somit zur `runWithResult`-Methode des `TransactionTemplates` zurück. Diese kann sich dann mit den die Transaktion abschließenden Aufgaben befassen (commit der Transaktion, close des `EntityManagers`). Und schließlich wird die `runWithResult`-Methode genau dasjenige Resultat zurückliefern, welches die `apply`-Methode von `SomeFunction` zurückgeliefert hat.

Das Interface `TransactionTemplate.Function<T>` könnte etwa wie folgt in einer anwendungsspezifischen Klasse implementiert sein:

```

public class Finder implements TransactionTemplate.Function<Book> {
    public Book apply(EntityManager manager) {
        return manager.find(Book.class, "1111");
    }
}

```

```
    }
}
```

Manchmal liefert ein anwendungsspezifischer Code aber überhaupt nichts zurück. Eine Function müsste dann `null` liefern – was natürlich unschön ist.

Statt `Function` kann dann das Interface `Consumer` verwendet werden:

```
public class TransactionTemplate {

    @FunctionalInterface
    public interface Consumer<T> {
        public abstract void accept(EntityManager manager);
    }
    // ...
}
```

Eine anwendungsspezifische Implementierung dieses Interfaces könnte wie folgt aussehen:

```
public class Persister implements TransactionTemplate.Consumer {
    public void accept(EntityManager manager) {
        manager.persist(new Book("1111", "Pascal", 10));
        manager.persist(new Book("2222", "Modula", 20));
        manager.persist(new Book("3333", "Oberon", 30));
    }
}
```

Die `TransactionTemplate`-Klasse besitzt neben der `runWithResult`-Methode eine einfache `run`-Methode, der ein `Consumer` übergeben wird.

Hier die vollständige `TransactionTemplate`-Klasse:

TransactionTemplate

```
package jpa.util;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;

public class TransactionTemplate {

    @FunctionalInterface
    public interface Consumer {
        public abstract void accept(EntityManager manager);
    }

    @FunctionalInterface
    public interface Function<T> {
        public abstract T apply(EntityManager manager);
    }
}
```

```
private final EntityManagerFactory factory;

public TransactionTemplate(EntityManagerFactory factory) {
    this.factory = factory;
}

public void run(Consumer callback) {
    EntityManager manager = this.factory.createEntityManager();
    EntityTransaction transaction = manager.getTransaction();
    try {
        transaction.begin();
        callback.accept(manager);
        transaction.commit();
    }
    catch (RuntimeException e) {
        this.log(e.toString());
        if (transaction.isActive()) {
            transaction.rollback();
        }
        throw e;
    }
    finally {
        manager.close();
    }
}

public <T> T runWithResult(Function<T> callback) {
    EntityManager manager = this.factory.createEntityManager();
    EntityTransaction transaction = manager.getTransaction();
    try {
        transaction.begin();
        T result = callback.apply(manager);
        transaction.commit();
        return result;
    }
    catch (RuntimeException e) {
        this.log(e.toString());
        if (transaction.isActive()) {
            transaction.rollback();
        }
        throw e;
    }
    finally {
        manager.close();
    }
}

public void runPerformanceTest(
    int loopCount, Consumer... callbacks) { ... }
}
```

(Die `run`-Methode könnte natürlich auch auf den Aufruf der `runWithResult`-Methode zurückgeführt werden...)

Die `runWithResult`-Methode könnte nun etwa mit einem `Finder` (s.o.) aufgerufen werden:

```
TransactionTemplate tt = new TransactionTemplate(factory);
Book book = tt.runWithResult("find", new Finder());
```

Die `run`-Methode könnte mit einem `Persister` (s.o.) aufgerufen werden:

```
tt.run(new Persister());
```

Natürlich könnte man die Interfaces `Consumer` resp. `Function` auch mittels anonymer Klassen implementieren – oder besser noch: in Form von Lambdas (denn die Interfaces sind "funktionale Interfaces" – Interfaces mit jeweils genau einer einzigen abstrakten Methode). Genau diese letzte Implementierungs-Variante wird in allen folgenden Beispielen genutzt.

Hier eine mögliche Applikation:

Application

```
package appl;
// ...
import jpa.util.TransactionTemplate;

public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();
        final EntityManagerFactory factory =
            Configuration.getEntityManagerFactory();
        try {
            TransactionTemplate tt = new TransactionTemplate(factory);
            demoPersist(tt);
            demoFind(tt);
            demoRemoveUpdate(tt);
            demoQuery(tt);
        }
        finally {
            factory.close();
        }
    }

    static void demoPersist(TransactionTemplate tt) {
        Util.mlog();
        tt.run(manager -> {
            manager.persist(new Book("1111", "Pascal", 10));
            manager.persist(new Book("2222", "Modula", 20));
            manager.persist(new Book("3333", "Oberon", 30));
        });
    }

    static void demoFind(TransactionTemplate tt) {
```



```
        Util.mlog();
        final Book book = tt.runWithResult(
            manager -> manager.find(Book.class, "2222"));
        System.out.println(book);
    }

    static void demoRemoveUpdate(TransactionTemplate tt) {
        Util.mlog();
        tt.run(manager -> {
            final Book book1 = manager.find(Book.class, "1111");
            manager.remove(book1);
            final Book book2 = manager.find(Book.class, "2222");
            book2.setTitle("Modula-2");
        });
    }

    static void demoQuery(TransactionTemplate tt) {
        Util.mlog();
        final List<Book> bookList = tt.runWithResult(manager -> {
            final TypedQuery<Book> query = manager.createQuery(
                "select b from Book b", Book.class);
            return query.getResultList();
        });
        bookList.forEach(System.out::println);
    }
}
```

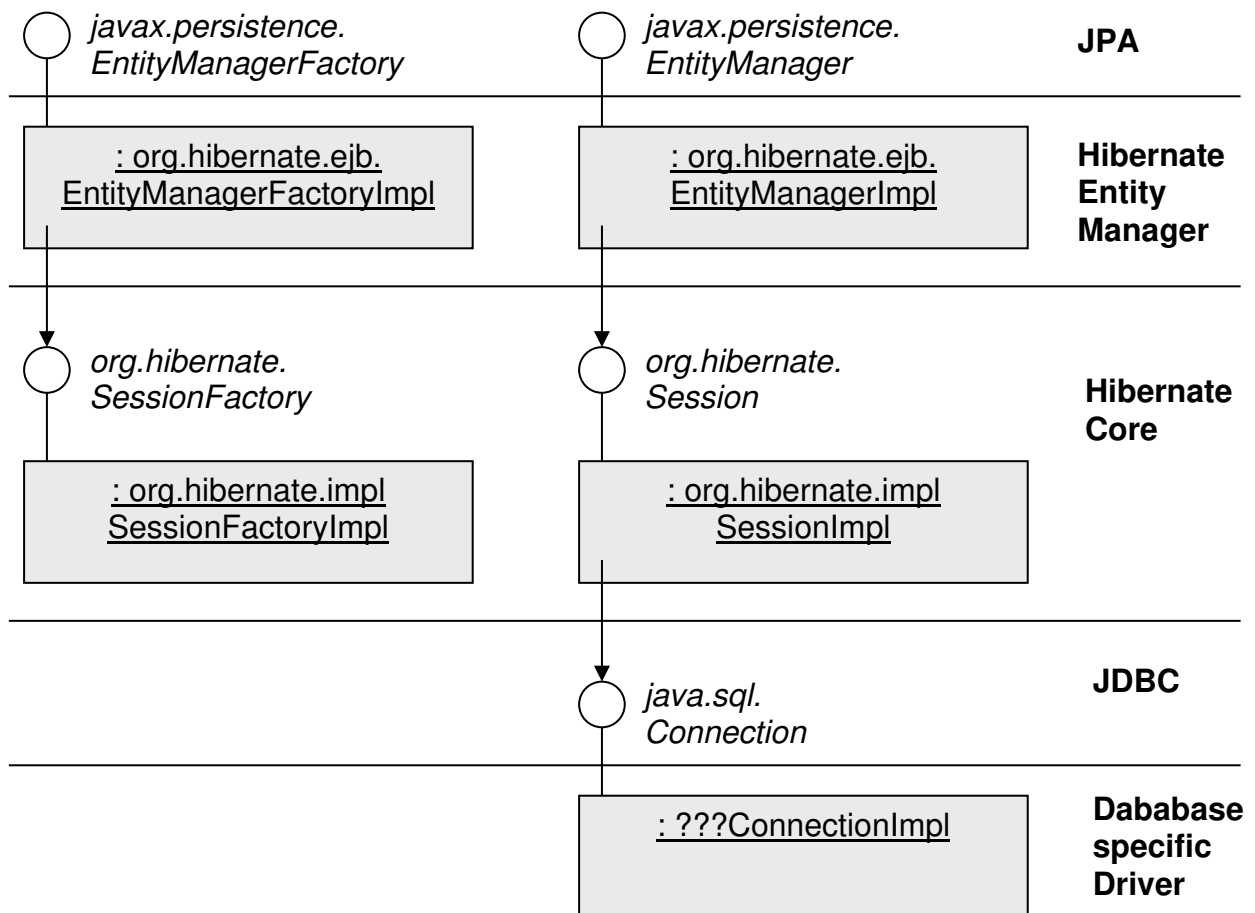
Man sieht: die eigentliche Anwendung ist wesentlich übersichtlicher geworden. Und man hat die Gewähr, dass alle Transaktionen nach demselben Muster ablaufen.

3.5 JPA und Hibernate

Im folgenden wird der Zusammenhang zwischen JPA und Hibernate näher erläutert.

JPA ist im Grunde nichts als eine Sammlung von Interfaces (und: einigen Enums und Annotations). Hibernate enthält eine Implementierung dieser Interfaces, welche diese auf den Hibernate-Core abbildet.

Das folgende Schaubild soll diese Zusammenhänge verdeutlichen:



JPA definiert die beiden zentralen Interfaces `EntityManagerFactory` und `EntityManager`. Diese werden u.a. von Hibernate implementiert – die Implementierungsklassen heißen `EntityManagerFactoryImpl` resp. `EntityManagerImpl`. Diese Klassen gehören zur JPA-Implementierung von Hibernate ("Hibernate Entity Manager") – sie gehören nicht zum Kern von Hibernate. Diese Implementierungsklassen nutzen Interfaces des Kerns von Hibernate: `SessionFactory` resp. `Session`. Diese Interfaces wiederum sind implementiert in den Klassen

`SessionFactoryImpl` und `SessionFactory`. Ein `SessionImpl`-Objekt bezieht sich stets auf eine Implementierung des JDBC-Interfaces `Connection` (die Implementierung ist i.d.R. Sache des Datenbank-anbieters und gehört zu demjenigen Komplex, der gewöhnlich als JDBC-Driver bezeichnet wird).

Was also auf der Ebene von JPA als `EntityManagerFactory` resp. `EntityManager` bezeichnet wird, entspricht auf der Hibernate-Ebene den Interfaces `SessionFactory` und `Session`.

So delegiert z.B. die `persist`-Methode von `EntityManager` (genauer: `EntityManagerImpl`) an die `save`-Methode von `Session` (genauer: von `SessionImpl`).

Die folgende Anwendung demonstriert diese Zusammenhänge. Sie zeigt u.a., wie mittels der `EntityManager`-Methode `getDelegate` der "JPA-Provider" (also das `Session`-Objekt von Hibernate) ermittelt werden kann.

Application

```
package appl;

// ...
import jpa.util.HibernateCache;

import org.hibernate.Session;

public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();
        final EntityManagerFactory factory =
            Configuration.getEntityManagerFactory();
        try {
            demo(factory);
        }
        finally {
            factory.close();
        }
    }

    static void demo(EntityManagerFactory factory) {
        Util.mlog();
        System.out.println("factory.class.name = " +
            factory.getClass().getName());
        final EntityManager manager = factory.createEntityManager();
        System.out.println("manager.class.name = " +
            manager.getClass().getName());
        final Session session = (Session) manager.getDelegate();
        System.out.println("session.class.name = " +
            session.getClass().getName());
        final EntityTransaction transaction = manager.getTransaction();
        System.out.println("transaction.class.name = " +
```

```

        transaction.getClass().getName());
    try {
        transaction.begin();
        final Book book = new Book("1111", "Pascal", 10);
        manager.persist(book);
        book.setPrice(2000);
        manager.persist(new Book("2222", "Modula", 20));
        session.save(new Book("3333", "Oberon", 30));
        transaction.commit();
    }
    catch (final RuntimeException e) {
        System.out.println(e);
        if (transaction.isActive())
            transaction.rollback();
        throw e;
    }
    finally {
        manager.close();
    }
}
}

```

Hier die Ausgaben:

```

+-----+
| demo  |
+-----+
factory.class.name =
org.hibernate.jpa.internal.EntityManagerFactoryImpl
manager.class.name = org.hibernate.jpa.internal.EntityManagerImpl
session.class.name = org.hibernate.internal.SessionImpl
transaction.class.name = org.hibernate.jpa.internal.TransactionImpl
Hibernate: insert into BOOK (PRICE, TITLE, ISBN) values (?, ?, ?)
Hibernate: insert into BOOK (PRICE, TITLE, ISBN) values (?, ?, ?)
Hibernate: insert into BOOK (PRICE, TITLE, ISBN) values (?, ?, ?)
Hibernate: update BOOK set PRICE=?, TITLE=? where ISBN=?

```

Die Klasse `Persistence` ist die einzige "richtige" Klasse, die auf der JPA-Ebene implementiert ist (alle anderen Klassen dieser Ebene repräsentieren Exceptions, Enums und Annotations). Ansonsten enthält JPA, wie oben bereits erwähnt, nur Interfaces. Die Klasse `Persistence` dient mit ihrer Methode `createEntityManagerFactory` als "Einstiegs-Klasse" (ähnlich wie die `DriverManager`-Klasse mit ihrer `getConnection`-Methode auf der JDBC-Ebene).

Abschließend einige weitere, wichtige Hinweise:

Die `EntityManagerFactory` ist thread-safe (sie ist ein konstantes Objekt); ein `EntityManager` ist nicht thread-safe. Es gibt stets eine einzige, "globale" `EntityManagerFactory`. Aber für jeden Thread wird es einen eigenen `EntityManager` geben müssen. Denn ein `EntityManager` ist mit einer `Connection` verbunden...

Die `EntityManagerFactory` (genauer: ein `SessionFactoryImpl`-Objekt) verwaltet u.a. einen `Connection-Pool`. Immer dann, wenn mittels der `Factory` ein `EntityManager` (also: ein `SessionImpl`-Objekt) erzeugt wird, wird dem `Pool` eine augenblicklich unbenutzte `Connection` entnommen und dem `EntityManager` exklusiv zur Verfügung gestellt. Die `close`-Methode, welche auf den `EntityManager` aufgerufen wird, stellt u.a. diese `Connection` wieder in den `Factory`-eigenen `Pool` zurück. Der Vorteil dieses `Connection-Poolings` besteht darin, dass beim Erzeugen eines neuen `EntityManagers` nicht immer eine neue `Connection` aufgebaut werden muss (was aufwendig ist) und beim `close` diese `Connection` wieder abgebaut werden muss.

Ein `EntityManager` (eine `Session`) sollte auf eine einzige "unit of work" beschränkt sein. Er ist also stets "kurzlebig". Daher ist auch die in den bisherigen Abschnitten benutzte Verknüpfung der Lebenszeit eines `EntityManager` mit derjenigen einer Transaktion sinnvoll. Während einer solchen Transaktion sollte die Anwendung insbesondere nicht auf Benutzereingaben warten. Bezüglich einer Web-Anwendung heißt dies, dass pro `Request` ein neuer `EntityManager` erzeugt wird und dieser nach Abschluss der Bearbeitung des `Requests` wieder geschlossen wird (noch bevor der `Request` an die JSP-Seite weitergeleitet wird!). Eine Lebensdauer einer Hibernate-`Session` sollte also auf gar keinen Fall an die `HttpSession` geknüpft sein – letztere lebt viel zu lange (umfasst i.d.R. viele `Request-/Response`-Zyklen). (Der Hibernate-Begriff `Session` ist in dieser Hinsicht leicht irreführend – er lädt zu Assoziationen zur `HttpSession` ein).

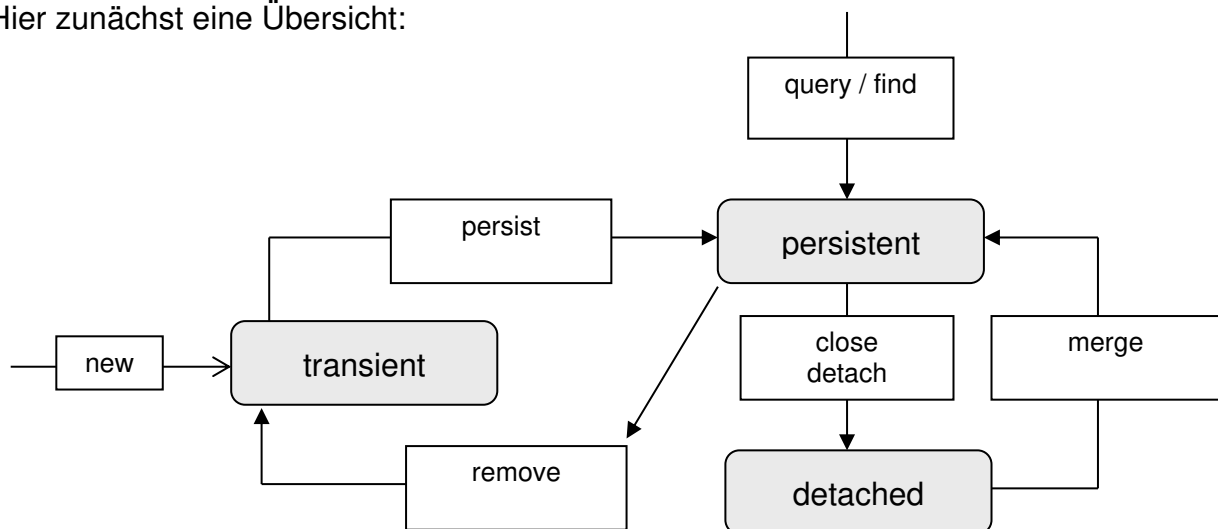
3.6 Der Cache und die Objekt-Stati

Ein `EntityManager` ist (via `SessionImpl`) mit einem Cache verbunden. Dieser Cache hat dieselbe Lebensdauer wie der `EntityManager` selbst: beim Erzeugen eines `EntityManagers` wird der Cache erzeugt, beim Schließen des `EntityManagers` wird er verworfen.

Mit diesem Cache ist der Begriff "Zustand eines Entity-Objekts" eng verbunden.

Ein Objekt einer persistenten Klasse kann sich aus Sicht von JPA in verschiedenen Zuständen befinden.

Hier zunächst eine Übersicht:



- Ein Objekt ist **transient**, wenn es per `new` erzeugt wurde und dieses Objekt noch nie mit JPA zu tun hatte.
- Ein Objekt ist **persistent**, wenn es sich im Cache eines aktiven `EntityManager`s befindet (es muss noch nicht per `INSERT` in die Datenbank übernommen worden sein!)
- Ein Objekt ist **detached**, wenn es zuvor im Cache eines `EntityManager` enthalten war und der `EntityManager` mittels `close` geschlossen wurde.

In den folgenden Abschnitten geht's um die möglichen Übergänge zwischen diesen Zuständen.

Anstatt den kompletten Programmcode darzustellen, werden im folgenden nur die einzelnen Transaktionen dargestellt. Zusammen mit dem Code der Transaktionen werden die erzeugten Ausgaben dargestellt – und der jeweils aktuelle Zustand der

Datenbank (am Ende jeder der im folgenden dargestellten Transaktionen wird eine weitere Transaktion gestartet, die den aktuellen Zustand der Datenbank ausgibt – diese Transaktion ist im folgenden aber nicht dargestellt (s. den kompletten Programmcode)).

Um den Cache etwas näher analysieren zu können, wird eine Klasse des jpa-util-Projekts verwendet: die Klasse `HibernateCache`. Diese besitzt eine statische Methode namens `printContext`, welche den aktuellen Zustand eines Caches eines `EntityManagers` ausgibt.

Im folgenden werden einige der wichtigsten Methoden des `EntityManagers` (und deren Auswirkungen auf den Cache und den Zustand der Objekte) näher beleuchtet werden:

- `persist`
- `find`
- `contains`
- `merge`
- `refresh`
- `remove`
- `detach`

Zunächst werden folgende `Book`-Objekte erzeugt, auf welche in den anschließenden Transaktionen zugegriffen wird (und die über statische Variablen angesprochen werden können):

```
static final Book book1 = new Book("1111", "Pascal", 10);
static final Book book2 = new Book("2222", "Modula", 20);
static final Book book3 = new Book("3333", "Oberon", 30);
```

Nun zu den einzelnen Transaktionen.

demoPersist

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        System.out.println(manager.contains(book1));
        manager.persist(book1);
        manager.persist(book2);
        manager.persist(book3);
        final Book b = manager.find(Book.class, "1111");
        System.out.println(b == book1);
        b.setTitle("Pascal-2");
        System.out.println(manager.contains(b));
        HibernateCache.printContext(manager);
    });
}
```

Die Ausgaben:

```
false
```

```

true
true
org.hibernate.internal.SessionImpl
  [ Dirty = true ]
  1111 ==>
    Book [1111, 10.0, Pascal-2]
    LoadedState = [ 10.0 Pascal ]
  2222 ==>
    Book [2222, 20.0, Modula]
    LoadedState = [ 20.0 Modula ]
  3333 ==>
    Book [3333, 30.0, Oberon]
    LoadedState = [ 30.0 Oberon ]

Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
Hibernate: update Book set price=?, title=? where isbn=?

```

Der Zustand der Datenbank:

```

domain.Book [1111, Pascal-2, 10.0]
domain.Book [2222, Modula, 20.0]
domain.Book [3333, Oberon, 30.0]

```

Anfangs ist `book1` noch nicht im Cache enthalten (`contains` liefert `false`). Die `persist`-Aufrufe tragen die drei übergebenen `Book`-Objekte im Cache ein (der `INSERT` findet hier aber noch nicht statt!). Die Objekte, die zuvor "transient" werden somit "persistent" (aber noch nicht "persistent" im Sinne der Datenbank!).

Die `find`-Methode versucht offenbar zunächst, im Cache fündig zu werden (wäre ein Objekt mit dem verlangten Schlüssel aber nicht im Cache enthalten, würde ein `SELECT` erfolgen). Das Ergebnis von `find` ist identisch mit `book1`. Es wird also genau dasjenige Objekt zurückgeliefert, welches zuvor mittels `persist` in den Cache eingetragen wurde.

Dann wird der Titel des `Book`-Objekt mittels `setTitle` geändert. Vom Aufruf dieser Methode hat Hibernate natürlich keine Ahnung!

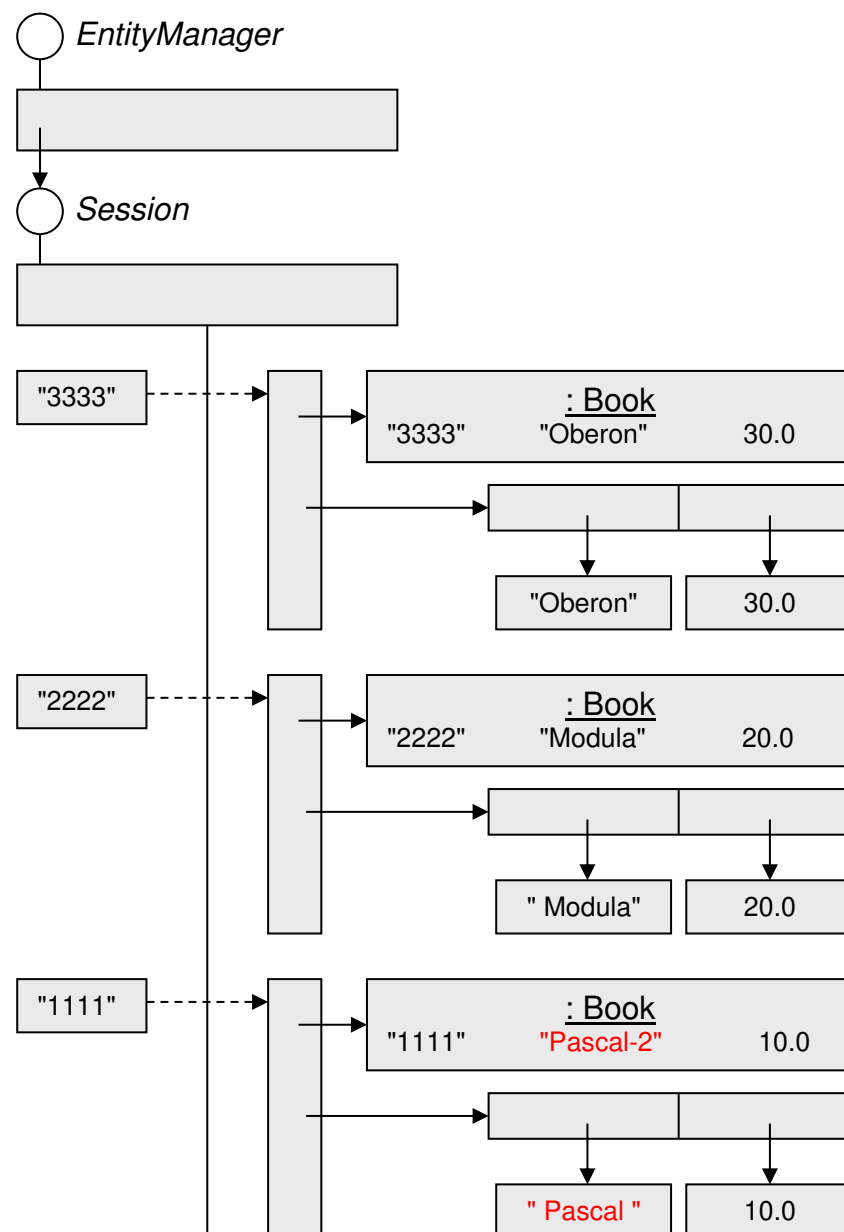
Beim `commit` erst werden die im Cache eingetragenen Objekte per `INSERT` in die Datenbank eingetragen. Und es findet ein zusätzlicher `UPDATE` statt.

Dieser `UPDATE` ist nur deshalb möglich, weil immer dann, wenn ein Objekt (wie auch immer!) den Cache betritt, nicht nur dieses Objekt gespeichert wird, sondern auch dessen "Loaded State". Dieser Loaded State besteht aus denjenigen Werten, die das Objekt beim Einfügen in den Cache besitzt (nur der Primary Key muss nicht noch einmal gespeichert werden – dieser dient im Cache vielmehr als Key, der es erlaubt, die Objekte schnell aufzufinden). Siehe die Ausgabe der Methode `printContext`!

Beim `commit` wird nun zunächst erkannt, dass der Cache drei Objekte enthält, die noch nicht mittels `INSERT` in die Datenbank eingefügt wurden. Also werden diese `INSERTs` nachgeholt. Schließlich wird erkannt, dass der Loaded State des "Pascal"-Buches sich vom aktuellen Zustand des entsprechenden `Book`-Objekts unterscheidet – und eben deshalb wird ein zusätzlicher `UPDATE` ausgeführt.

Nach dem `close` des `EntityManager`s sind die drei `Book`-Objekte "detached" – sie sind von Hibernate verlassen (gleichwohl sind sie im Sinne der Datenbank natürlich "persistent").

Hier der Zustand des Caches vor dem `commit`:



Hinweis: Wir setzen hierbei voraus, dass pro Programmstart immer nur eine der folgenden Demo-Methoden aufgerufen wird!

demoMerge1

```
static void demoMerge1(TransactionTemplate tt) {
    tt.run(manager -> {
        System.out.println(manager.contains(book2));
        final Book b = manager.merge(book2);
        System.out.println(b == book2);
        System.out.println(manager.contains(book2));
        System.out.println(manager.contains(b));
        book2.setTitle("Modula-2");
        HibernateCache.printContext(manager);
    });
}
```

Die Ausgaben:

```
false
Hibernate: select ... from Book b where b.isbn=?
false
false
true
org.hibernate.internal.SessionImpl
[ Dirty = false ]
2222 ==>
    Book [2222, 20.0, Modula]
    LoadedState = [ 20.0 Modula ]
```

Der Zustand der Datenbank:

```
1111 Pascal-2 10.0
2222 Modula 20.0
3333 Oberon 30.0
```

Beim Eintritt in diese Transaktion ist das über `book2` referenzierte Objekt "detached".

Der Aufruf der `merge`-Methode bewirkt folgendes: Der Primary Key des übergebenen Objekts (hier: die ISBN-Nummer des übergebenen `Books`) wird genutzt, um einen `SELECT` auszuführen. Dieser `SELECT` liefert genau eine Zeile (wenn alles gut geht...). Aufgrund diese Zeile erzeugt Hibernate ein neues(!) `Book` und initialisiert dieses mit den Daten der aktuell gelesenen Zeile.

Dieses (neue!) `Book` wird im Cache abgestellt (das `Book` und der aktuelle Loaded State). Schließlich wird der Zustand des an `merge` übergebenen Objekts in das neu erzeugte

Objekt übertragen. Das neu erzeugte `Book` wird als Ergebnis von `merge` zurückgeliefert. Und dieses neue `Book` ist nun "persistent" – nicht aber das an `merge` übergebene!

Anhand der Ausgaben wird deutlich, dass das an `merge` übergebene `Book` nicht dasjenige `Book` ist, was `merge` zurückliefert. Der Cache kennt das übergebene Objekt überhaupt nicht (sondern nur das neu erzeugte) – das zeigen die beiden `contains`-Aufrufe. Anschließende Änderungen auf das an `merge` übergebene Objekt bleiben somit ohne "durchschlagende" Wirkung: es wird kein `UPDATE` ausgeführt.

demoMerge2

```
static void demoMerge2(TransactionTemplate tt) {
    tt.run(manager -> {
        System.out.println(manager.contains(book2));
        book2.setTitle("Modula-2");
        final Book b = manager.merge(book2);
        System.out.println(b == book2);
        System.out.println(manager.contains(book2));
        System.out.println(manager.contains(b));
        HibernateCache.printContext(manager);
    });
}
```

Die Ausgaben:

```
false
Hibernate: select book0_.isbn as isbn1_0_0_, book0_.price as
price2_0_0_, book0_.title as title3_0_0_ from Book book0_ where
book0_.isbn=?
false
false
true
org.hibernate.internal.SessionImpl
[ Dirty = true ]
2222 ==>
    Book [2222, 20.0, Modula-2]
    LoadedState = [ 20.0 Modula ]
```

```
Hibernate: update Book set price=?, title=? where isbn=?
```

Der Zustand der Datenbank:

```
1111 Pascal-2 10.0
2222 Modula-2 20.0
3333 Oberon 30.0
```

Hier wird nun auf `book2` die `setTitle`-Methode aufgerufen, bevor(!) dieses `Book` an `merge` übergeben wird. Dies führt dazu, dass der Titel des von `merge` neu erzeugten `Books` nach dem Eintrag in den Cache geändert wird (der Zustand des übergebenen

Books wird dem neu erzeugten `Book` zugewiesen). Am Ende der Transaktion wird dann erkannt, dass der Loaded State sich vom aktuellen Zustand des Objekts unterscheidet – und also findet ein `UPDATE` statt.

`merge` – so könnte man sagen – synchronisiert von "oben" (Java) nach "unten" (Datenbank). Aber nur dann, wenn man es richtig macht. Um Probleme wie beim `Refresh 1` zu vermeiden, sollte das Ergebnis von `merge` genau derjenigen Referenz zugewiesen werden, welche übergeben wird: `b = manager.merge(b)`.

demoRefresh1

```
static void demoRefresh1(TransactionTemplate tt) {
    tt.run(manager -> {
        System.out.println(manager.contains(book3));
        final Book b = manager.find(Book.class, "3333");
        b.setTitle("Oberon-2");
        manager.refresh(b);
        System.out.println(manager.contains(book3));
        System.out.println(manager.contains(b));
        System.out.println(b);
        HibernateCache.printContext(manager);
    });
}
```

Die Ausgaben:

```
false
Hibernate: select isbn, price, title from Book where isbn=?
Hibernate: select isbn, price, title from Book where isbn=?
false
true
Book [3333, 30.0, Oberon]
org.hibernate.internal.SessionImpl
  [ Dirty = false ]
  3333 ==>
    Book [3333, 30.0, Oberon]
    LoadedState = [ 30.0 Oberon ]
```

Der Zustand der Datenbank:

```
1111 Pascal-2 10.0
2222 Modula 20.0
3333 Oberon 30.0
```

Per `find` wird ein neues `Book` erzeugt und im Cache abgestellt (es ist also anschließend "persistent"). Dann wird der Titel des `Books` geändert. Also unterscheiden sich nun aktueller Zustand und Loaded State. Würde man es hierbei belassen und den Manager schließen, so würde ein `UPDATE` ausgeführt werden.

Aber es findet ja noch ein `refresh`-Aufruf statt. An `refresh` muss ein "persistentes" Objekt übergeben werden (daher zuvor der `find`-Aufruf!). Der Aufruf von `refresh` bewirkt folgendes: Es wird erneut auf die Datenbank zugegriffen. Die Daten der aktuellen Zeile werden in das an `refresh` übergebene persistente Objekt eingetragen. Und somit ist der aktuelle Zustand wieder gleich dem aufgrund des anfänglichen `finds` gespeicherten Loaded State. Daher findet am Ende auch kein `UPDATE` statt. (Man beachte, dass `refresh` im Gegensatz zu `merge` kein neues Objekt erzeugt und daher auch vom Typ `void` ist.)

demoRefresh2

```
static void demoRefresh2(TransactionTemplate tt) {
    tt.run(manager -> {
        System.out.println(manager.contains(book3));
        final Book b = manager.find(Book.class, "3333");
        manager.refresh(b);
        b.setTitle("Oberon-2");
        System.out.println(manager.contains(book3));
        System.out.println(manager.contains(b));
        System.out.println(b);
        HibernateCache.printContext(manager);
    });
}
```

Die Ausgaben:

```
false
Hibernate: select isbn, price, title from Book where isbn=?
Hibernate: select isbn, price, title from Book where isbn=?
false
true
Book [3333, 30.0, Oberon-2]
org.hibernate.internal.SessionImpl
[ Dirty = true ]
3333 ==>
    Book [3333, 30.0, Oberon-2]
    LoadedState = [ 30.0 Oberon ]
```

```
Hibernate: update Book set price=?, title=? where isbn=?
```

Der Zustand der Datenbank:

```
1111 Pascal-2 10.0
2222 Modula 20.0
3333 Oberon-2 30.0
```

Hier wird der Titel erst nach dem `refresh` neu gesetzt (der `refresh` hatte also keinerlei Netto-Effekt). Loaded State und aktueller Zustand unterscheiden sich also beim Ende der Transaktion. Also findet ein `UPDATE` statt.

Im Gegensatz zum `merge` handelt es sich beim `refresh` also um eine Synchronisation von "unten" (Datenbank) nach "oben" (Java).

demoRemove

```
static void demoRemove(TransactionTemplate tt) {
    Util.mlog();
    tt.run(manager -> {
        final Book b = manager.find(Book.class, "2222");
        System.out.println(manager.contains(b));
        manager.remove(b);
        System.out.println(manager.contains(b));
        HibernateCache.printContext(manager);
    });
}
```

Die Ausgaben:

```
Hibernate: select isbn, price, title from Book where isbn=?
true
false
org.hibernate.internal.SessionImpl
  [ Dirty = true ]
  2222 ==>
    Book [2222, 20.0, Modula]
    LoadedState = [ 20.0 Modula ]
    DeletedState = [ 20.0 Modula ]
```

```
Hibernate: delete from Book where isbn=?
```

Der Zustand der Datenbank:

```
1111 Pascal-2 10.0
3333 Oberon 30.0
```

An `remove` muss ebenso wie beim `merge` und beim `refresh` ein persistentes Objekt übergeben werden (eines, das im Cache enthalten ist). Deshalb der anfängliche Aufruf von `find`. Nach dem Aufruf von `remove` ist das Objekt (logisch) aus dem Cache verschwunden (`contains` liefert dann `false`). Tatsächlich ist es noch im Cache enthalten – aber als zu löschendes Objekt markiert (s. die Ausgabe von `printContext`). Beim `commit` wird deshalb ein `DELETE` abgesetzt.

demoDetach

```
static void demoDetach(TransactionTemplate tt) {
    tt.run(manager -> {
        final Book b = manager.find(Book.class, "3333");
        System.out.println(manager.contains(b));
    });
}
```

```

        manager.detach(b);
        System.out.println(manager.contains(b));
        book3.setTitle("Oberon-3");
        HibernateCache.printContext(manager);
    });
}

```

Die Ausgaben:

```

Hibernate: select isbn, price, title from Book where isbn=?
true
false
org.hibernate.internal.SessionImpl
    [ Dirty = false ]

```

Der Zustand der Datenbank:

```

1111 Pascal-2 10.0
2222 Modula 20.0
3333 Oberon 30.0

```

Auch an `detach` muss ein persistentes Objekt übergeben werden. Per `detach` wird dieses Objekt (tatsächlich!) aus dem Cache entfernt – es wird also detached. Anschließende Änderungen an diesem Objekt haben daher keine durchschlagende Wirkung.

Hier abschließend der Quellcode der im obigen Beispiel verwendeten Klasse `HibernateCache` (dessen Analyse dem Leser überlassen bleiben soll):

HibernateCache

```

package jpa.util;

import java.util.Map.Entry;

import javax.persistence.EntityManager;

import org.hibernate.Session;
import org.hibernate.engine.spi.EntityEntry;
import org.hibernate.engine.spi.PersistenceContext;
import org.hibernate.internal.SessionImpl;

public class HibernateCache {

    public static void printContext(EntityManager manager) {
        Session session = (Session)manager.getDelegate();
        StringBuilder builder = new StringBuilder();
        builder.append(session.getClass().getName()).append("\n");
        builder.append("\t[ Dirty = " + session.isDirty() + " ]\n");
        SessionImpl impl = (SessionImpl) session;
    }
}

```

```
PersistenceContext ctx = impl.getPersistenceContext();
Entry<Object, EntityEntry>[] list = ctx.reentrantSafeEntityEntries();
for (Entry<Object, EntityEntry> e : list) {
    Object obj = e.getKey();
    EntityEntry entry = e.getValue();
    builder.append("\t" + entry.getId() + " ==> \n");
    builder.append("\t\t" + obj + "\n");
    Object[] loadedState = entry.getLoadedState();
    if (loadedState != null) {
        builder.append("\t\tLoadedState = [");
        for (Object value : loadedState)
            builder.append(" " + value);
        builder.append(" ]\n");
    }
    Object[] deletedState = entry.getDeletedState();
    if (deletedState != null) {
        builder.append("\t\tDeletedState = [");
        for (Object value : deletedState)
            builder.append(" " + value);
        builder.append(" ]\n");
    }
}
System.out.println(builder);
}
```


3.7 Callbacks

JPA kann angewiesen werden, die Objekte der persistenten Klassen über bestimmte Ereignisse zu informieren. Solche Objekte können z.B. darüber benachrichtigt werden, dass sie zuvor gerade vom JPA-Provider erzeugt wurden; dass ihr Inhalt in die Datenbank geschrieben wird; dass die entsprechende Tabellenzeile gelöscht wird etc.

Eine persistente Klasse kann mit Methoden ausgestattet werden, die mit folgenden Annotations gekennzeichnet sind:

```
@PrePersist  
@PostPersist
```

```
@PreUpdate  
@PostUpdate
```

```
@PreRemove  
@PostRemove
```

```
@PostLoad
```

Die Methoden müssen parameterlos und `void` sein.

Solche Methoden werden dann automatisch vom JPA-Provider aufgerufen, wenn die entsprechenden Ereignisse anstehen. Bevor also ein `INSERT` in eine Tabelle stattfindet, wird z.B. die mit `@PrePersist` gekennzeichnete Methode aufgerufen; nachdem der `INSERT` stattgefunden hat, wird `@PostPersist` aufgerufen.

Man beachte, dass es zwar `@PostLoad`, nicht aber `@PreLoad` gibt!

Book

```
package domain;  
// ...  
@Entity  
public class Book {  
  
    // wie gehabt...  
  
    @PrePersist  
    public void beforeInsert() {  
        System.out.println("PRE-PERSIST: " + this);  
    }  
    @PostPersist  
    public void afterInsert() {  
        System.out.println("POST-PERSIST: " + this);  
    }  
}
```

```

@PreUpdate
public void beforeUpdate() {
    System.out.println("PRE-UPDATE: " + this);
}
@PostUpdate
public void afterUpdate() {
    System.out.println("POST-UPDATE: " + this);
}

@PreRemove
public void beforeDelete() {
    System.out.println("PRE-REMOVE: " + this);
}
@PostRemove
public void afterDelete() {
    System.out.println("POST-REMOVE: " + this);
}

@PostLoad
public void afterLoad() {
    System.out.println("POST-LOAD: " + this);
}
}

```

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 30));
        manager.persist(new Book("2222", "Modula", 40));
    });
}

```

Die Ausgaben:

```

PRE-PERSIST: Book [1111, 30.0, Pascal]
PRE-PERSIST: Book [2222, 40.0, Modula]
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
POST-PERSIST: Book [1111, 30.0, Pascal]
Hibernate: insert into Book (price, title, isbn) values (?, ?, ?)
POST-PERSIST: Book [2222, 40.0, Modula]

```

```

static void demoRemove(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.remove(manager.find(Book.class, "2222"));
    });
}

```

Die Ausgaben:

```

Hibernate: select ... from Book isbn=?
POST-LOAD: Book [2222, 40.0, Modula]
PRE-REMOVE: Book [2222, 40.0, Modula]

```

Hibernate: delete from Book where isbn=?
POST-REMOVE: Book [2222, 40.0, Modula]

```
static void demoFind(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final Book book = manager.find(Book.class, "1111");  
        book.setTitle("Oberon");  
    });  
}
```

Die Ausgaben:

Hibernate: select ... from Book isbn=?
POST-LOAD: Book [1111, 30.0, Pascal]
PRE-UPDATE: Book [1111, 30.0, Oberon]
Hibernate: update Book set price=?, title=? where isbn=?
POST-UPDATE: Book [1111, 30.0, Oberon]

3.8 Listeners

Die oben in der persistenten Klasse implementierten "Callback"-Methoden können auch in einer separaten Klasse definiert werden (hier als "Listener"-Klasse bezeichnet). Dort müssen sie allerdings einen `Object`-Parameter besitzen - beim Aufruf wird ihnen nämlich das jeweilige persistente Objekt übergeben, dem das Ereignis gilt.

Die persistente Klasse selbst kann dann mit einer `@EntityListener`-Annotation gekennzeichnet werden. Dieser Annotation können beliebig viele `Class`-Referenzen übergeben werden - Referenzen auf `Class`-Objekte, die Listener-Klassen beschreiben. Der JPA-Provider wird dann diese Klassen instanziiieren. Und immer dann, wenn Objekte dieser so gekennzeichneten persistenten Klasse vom JPA-Provider manipuliert werden, werden die entsprechenden Callback-Methoden auf die Listener-Objekte aufgerufen.

Man beachte, dass die Listener-Klassen kein Interface implementieren - sondern einfach nur ihre Methode über entsprechende Annotations kennzeichnen.

Book

```
package domain;
// ...
@Entity
@EntityListeners({util.Logger.class})
public class Book {
    // ... OHNE Callback-Methoden
}
```

Logger

```
package appl;
// ...
public class Logger {

    @PrePersist
    public void beforeInsert(Object obj) {
        System.out.println("PRE-PERSIST: " + obj);
    }

    @PostPersist
    public void afterInsert(Object obj) {
        System.out.println("POST-PERSIST: " + obj);
    }

    @PreUpdate
    public void beforeUpdate(Object obj) {
        System.out.println("PRE-UPDATE: " + obj);
    }
}
```

```
@PostUpdate
public void afterUpdate(Object obj) {
    System.out.println("POST-UPDATE: " + obj);
}

@PreRemove
public void beforeDelete(Object obj) {
    System.out.println("PRE-REMOVE: " + obj);
}

@PostRemove
public void afterDelete(Object obj) {
    System.out.println("POST-REMOVE: " + obj);
}

@PostLoad
public void afterLoad(Object obj) {
    System.out.println("POST-LOAD: " + obj);
}
}
```

Es wird dieselbe Application genutzt wie im letzten Abschnitt.

Und auch die Ausgaben sind exakt identisch mit denjenigen des letzten Abschnitts.

3.9 Eine Basisklasse BaseEntity

In vielen Datenbanken enthalten alle Tabellen neben den eigentlichen Nutzdaten-Spalten zwei weitere Spalten: `USER` und `LASTUPDATE`. Die erste Spalte enthält den Benutzer, der die Zeile eingefügt resp. zuletzt geändert hat; die zweite Spalte den Zeitpunkt des Einfügens resp. des Änderns.

Die `BOOK`-Tabelle müsste dann wie folgt erweitert werden:

create.sql

```
create table BOOK (
    USERNAME varchar(32),
    LASTUPDATE timestamp,
    ISBN varchar (20),
    TITLE varchar (128) not null,
    PRICE double not null,
    primary key (ISBN)
);
```

Dementsprechend müsste auch die `Book`-Klasse um zwei Attribute mit entsprechenden Properties erweitert werden (`userName`, `lastUpdate`). Und jede andere persistente Klasse auch. Und man müsste an allen Stellen des Programmsystem, an denen auf Tabellen zugegriffen wird, immer für die korrekten Einträge von `userName` und `lastUpdate` sorgen – ein kleiner Alptraum. Das Setzen der entsprechenden Spaltenwerte kann einem Listener übertragen werden. Dieser Listener wird als `Stamper` bezeichnet werden.

Dann muss der `Stamper` aber den aktuellen Benutzer kennen (denn dieser kann dem `Stamper` nicht übergeben werden: der `Stamper` wird ja von Hibernate erzeugt werden!).

Also kann der aktuelle Benutzer an ein `TreadLocal` gebunden werden, welcher global ansprechbar ist (Singleton):

Users

```
package util;

public class Users extends ThreadLocal<String> {
    public static final Users instance = new Users();
    private Users() {
    }
}
```

Der Stamper besitzt eine `stamp`-Methode, welche mit `@PreUpdate` und `@PrePersist` annotiert ist. Sie wird also immer dann aufgerufen werden, wenn ein Datensatz eingefügt resp. verändert wird (und zwar vor dem Einfügen / Verändern):

Stamper

```
package util;
// ...
public class Stamper {

    @PreUpdate
    @PrePersist
    public void stamp(BaseEntity entity) {
        entity.setUsername(Users.instance.get());
        entity.setLastUpdate(new GregorianCalendar());
    }
}
```

Man beachte die Benutzung des `ThreadLocals`.

Anstatt nun in jeder `@Entity`-Klasse die beiden erforderlichen Attribute und Properties zu definieren, kann eine abstrakte Basisklasse implementiert werden: `BaseEntity`. Diese enthält die erforderlichen Attribute und Properties und kann auch zugleich per `@EntityListeners` veranlassen, dass Hibernate einen `Stamper` erzeugt (und diesen entsprechend benachrichtigen wird).

BaseEntity

```
package util;
// ...
@MappedSuperclass
@EntityListeners( { Stamper.class })
public abstract class BaseEntity {

    @Basic
    private String userName;

    @Basic
    private Calendar lastUpdate;

    // getter, setter...

    @Override
    public String toString() {
        return this.user + ", " +
            DateFormat.getInstance().format(this.lastUpdate);
    }
}
```

Man beachte die Verwendung von `@MappedSuperclass` – diese Annotation ist für abstrakte Basisklassen von `@Entity`-Klassen erforderlich.

Die Klasse `Book` kann dann auf einfache Weise von `BaseEntity` abgeleitet werden:

Book

```
package domain;
// ...
@Entity
public class Book extends BaseEntity {

    @Id
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    // Konstruktoren, getter, setter, toString...
}
```

Und hier schließlich die Beispielanwendung:

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 30));
        manager.persist(new Book("2222", "Modula", 40));
    });
}
```

Die Ausgaben:

```
Hibernate: insert into Book (...) values (?, ?, ?, ?, ?)
Hibernate: insert into Book (...) values (?, ?, ?, ?, ?)
```

Zwischen dem Aufruf von `demoPersist` und dem Aufruf der folgenden `demoRemove`-Methode schläft die Anwendung zwei Sekunden – um zeigen zu können, dass die Zeitstempel auch beim UPDATE gesetzt werden.

```
static void demoRemove(TransactionTemplate tt) {
    tt.run(manager -> {
        final Book book = manager.find(Book.class, "1111");
        book.setTitle("Oberon");
    });
}
```

Die Ausgaben:


```
Hibernate: select ... from Book where isbn=?
Hibernate: update Book set lastUpdate=?, userName=?, price=?, title=?
       where isbn=?
```

Die Datenbank sieht anschließend wie folgt aus:

```
BOOK
USERNAME LASTUPDATE          ISBN TITLE  PRICE
-----
Nowak     2018-03-09 17:57:36.213 1111 Oberon 30.0
Nowak     2018-03-09 17:57:34.132 2222 Modula 40.0
-----
```

3.10 Generierte Primary-Keys

Im folgenden geht's um generierte Primärschlüssel.

JPA unterstützt sowohl "Business-Keys" als auch von der Datenbank generierte Identifikatoren als Primary-Keys. Business-Keys haben eine eigene, anwendungsspezifische Semantik (z.B. ISBN-Nummer eines `BOOKS`). Generierte Schlüssel haben keine Semantik - sie haben nur die Funktion, Zeilen einer Tabelle eindeutig identifizieren zu können.

Sofern generierte Schlüssel als Primary-Keys benutzt werden, können gleichwohl zusätzlich Business-Keys definiert werden - mittels der `UNIQUE`-Klausel. Diese fungieren dann aber eben nicht mehr als `PRIMARY KEYS`.

Die Datenbank kann für die in die Tabellen einzufügenden Zeilen automatisch generierte Identifier vergeben. Diese Identifier können dann als Primary Keys verwendet werden. Das hat dann z.B. den entscheidenden Vorteil, dass keine zusammengesetzten Schlüssel erforderlich werden können - und auch die `FOREIGN KEYS` sind dann stets einfache Identifier-Werte.

Zur Generierung von solchen Identifier können unterschiedliche Strategien verwendet werden (s. weiter unten).

Soll z.B. Derby angewiesen werden, einen Identifier zu erzeugen, so wird z.B. die folgende Zeile verwendet (in einem `CREATE TABLE`):

```
ID integer generated by default as identity (start with 1),
```

Im folgenden wird die `BOOK`-Tabelle mit einem auf diese Weise generierten Primary Key ausgestattet. Die ISBN-Nummer dient weiterhin als Business-Key - sie wird in der Tabellendefinition als `unique` gekennzeichnet.

create.sql

```
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    ISBN varchar (20) not null,  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    primary key (ID),  
    unique (ISBN)  
);
```

Die Klasse `Book` definiert für die `ID`-Spalte ein entsprechendes Attribut und eine entsprechende Property. Attribut und Property müssen einen Referenztyp besitzen - z.B. `Integer`. Sie dürfen nicht vom Typ `int` sein. Der JPA-Provider muss nämlich

unterscheiden können, ob der Key bereits gesetzt wurde (und damit das Objekt bereits gespeichert wurde) oder nicht - das kann er aber nur dann, wenn das Attribut / die Property `null`-wertfähig (im Sinne der Java-`null`) sind.

Book

```
package domain;
// ...
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    Book() {
    }

    public Book(String isbn, String title, double price) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
    }

    // getter, setter, toString...
}
```

Dem Konstruktor wird weiterhin nur `isbn`, `title` und `price` übergeben. Wird also ein `Book`-Objekt von der Anwendung erzeugt, ist dessen `id` noch `null`. Erst dann, wenn der JPA-Provider das `Book` persistiert, wird das Feld ausgestattet mit der von der Datenbank vergebenen Identifier (per Aufruf von `setId`).

Das `id`-Attribut ist natürlich zunächst als `@Id` gekennzeichnet. Zusätzlich besitzt es eine weitere Annotation, die erstens ausdrückt, dass der Wert für `id` generiert wird und zweitens die gewünschte Generierungsstrategie benennt. Als Strategien können folgende Werte eingetragen werden:

```
IDENTITY
AUTO
SEQUENCE
TABLE
```

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        manager.persist(new Book("1111", "Pascal", 10));  
        manager.persist(new Book("2222", "Modula", 20));  
    });  
}
```

Die Ausgaben:

```
Hibernate: insert into Book (id, ...) values (default, ?, ?, ?)  
Hibernate: insert into Book (id, ...) values (default, ?, ?, ?)
```

Man beachte, dass die `INSERT`-Befehle unmittelbar beim Aufruf von `persist` abgesetzt werden. Denn die `persist`-Methode muss das ihr übergebene Objekt mit der von der Datenbank generierten ID ausstatten. Deshalb kann der `INSERT` nun nicht mehr auf den Zeitpunkt des Aufrufs von `commit` verschoben werden.

```
static void demoFind(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final Book book = manager.find(Book.class, 1);  
        System.out.println(book);  
    });  
}
```

Die Ausgaben:

```
Hibernate: select ... from Book where id=?  
Book [1, 1111, 10.0, Pascal]
```

Man beachte, dass die `find`-Methode nun eigentlich kaum noch sinnvoll verwendet werden kann (der Benutzer kennt keine IDs).

```
static void demoQuery(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final TypedQuery<Book> query = manager.createQuery(  
            "select b from Book b", Book.class);  
        final List<Book> books = query.getResultList();  
        books.forEach(System.out::println);  
    });  
}
```

Die Ausgaben:

```
Hibernate: select ... from Book  
Book [1, 1111, 10.0, Pascal]  
Book [2, 2222, 20.0, Modula]
```

Der Zustand der Datenbank:

BOOK

ID	ISBN	TITLE	PRICE
----	------	-------	-------

1	1111	Pascal	10.0
---	------	--------	------

2	2222	Modula	20.0
---	------	--------	------

3.11 Generierung mittels eines TableGenerators

Im folgenden wird eine zweite Schlüsselgenerierungsvariante vorgestellt, die `TABLE`-Strategie. Diese Variante basiert auf einer separaten Tabelle, welche für jede persistente Klasse einen bestimmten Startwert beinhaltet.

Sofern Hibernate als Persistence-Provider verwendet wird, sollte sinnvollerweise eine Tabelle namens `HIBERNATE_SEQUENCES` mit den Spalten `SEQUENCE_NAME` und `SEQUENCE_NEXT_HI_VALUE` verwendet werden - dann muss in den persistenten Klassen keine eigene Tabellenstruktur vereinbart werden.

Der in der Tabelle angegebene Startwert wird beim Erzeugen eines `EntityManagers` gelesen und jeweils um 1 inkrementiert. Der `EntityManager` hat dann einen bestimmten zusammenhängenden Bereich von `Ids` zur ausschließlichen Verfügung. Erst wenn alle Keys dieses Bereichs vergeben sind, muss der `EntityManager` sich einen weiteren Schlüsselvorrat besorgen (wobei dann der Startwert erneut inkrementiert wird).

Natürlich wird ein `EntityManager` i.d.R. nicht den gesamte Vorrat an zu vergebenden Schlüsseln ausnutzen - die generierten Schlüssel werden also i.d.R. große Lücken aufweisen. Aber generierte Schlüssel müssen nicht lückenlos sein, sie müssen nur eindeutig sein.

create.sql

```
create table SEQUENCE (
    SEQ_NAME varchar (255),
    SEQ_COUNT decimal (15),
    primary key (SEQ_NAME)
);

insert into SEQUENCE values ('BOOK', 0);

create table BOOK (
    ID integer,
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    primary key (ID),
    unique (ISBN)
);
```

Book

```
package domain;
// ...
@Entity
```

```

public class Book {

    @TableGenerator(
        name = "tableGenerator",
        table = "SEQUENCE",
        pkColumnName = "SEQ_NAME",
        pkColumnValue = "BOOK",
        valueColumnName = "SEQ_COUNT",
        allocationSize = 20)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
        generator = "tableGenerator")
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    // Konstruktoren, getter, setter, toString...
}

```

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
        manager.persist(new Book("2222", "Modula", 20));
        manager.persist(new Book("3333", "Oberon", 30));
        manager.persist(new Book("4444", "Eiffel", 40));
    });
}

```

Die Ausgaben:

```

Hibernate: select SEQ_COUNT from SEQUENCE where
    SEQ_NAME = 'BOOK' for update
Hibernate: update SEQUENCE set SEQ_COUNT = ?
    where SEQ_COUNT = ? and SEQ_NAME = 'BOOK'
Hibernate: insert into Book (...) values (?, ?, ?, ?)
Hibernate: insert into Book (...) values (?, ?, ?, ?)
Hibernate: insert into Book (...) values (?, ?, ?, ?)
Hibernate: insert into Book (...) values (?, ?, ?, ?)

```

```

static void demoFind(TransactionTemplate tt) {
    tt.run(manager -> {
        final Book book = manager.find(Book.class, 1);
        System.out.println(book);
    });
}

```

Die Ausgaben:

```
Hibernate: select ... from Book where id=?  
Book [1, 1111, 10.0, Pascal]
```

Das Resultat in der Datenbank:

```
SEQUENCE  
SEQ_NAME SEQ_COUNT  
-----  
BOOK      1  
-----
```

```
BOOK  
ID ISBN TITLE  PRICE  
-----  
1  1111 Pascal  10.0  
2  2222 Modula  20.0  
3  3333 Oberon  30.0  
4  4444 Eiffel  40.0  
-----
```


3.12 FindByBusinessKey

Angenommen, alle Tabellen besitzen generierte Schlüssel. Für jede Tabelle kann dann natürlich weiterhin ein Business-Key definiert sein - in Form einer `UNIQUE`-Klausel.

Die `EntityManager`-Methode `find` verlangt als Argument den Primary-Key. Dieser ist bei generierten Schlüsseln natürlich i.d.R. nicht bekannt. Also muss über den Business-Key zugegriffen werden können - das aber ist eben mit `find` nicht möglich. Man benötigt also für den Zugriff per Business-Key einen "komplexen" Query.

Es wäre schön, wenn man eine Methode hätte, welche diese Komplexität kapselt. Diese Methode existiert in der Klasse `QueryUtil` (im `jap-util`-Projekt). Bevor diese Methode vorgestellt wird, hier zunächst eine Anwendung dieser Methode:

create.sql

```
create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    primary key (ID),
    unique (ISBN)
);
```

Bei der `BOOK`-Tabelle wird eine generierte `ID` als Primary Key verwendet. In `BOOK` ist aber zusätzlich die `ISBN`-Spalte als `UNIQUE` gekennzeichnet. `ISBN` spielt dort also die Rolle eines Business Keys.

Book

```
package domain;
// ...
@Entity
@Table(uniqueConstraints=@UniqueConstraint(columnNames={"isbn"}))

public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
```

```
private double price;  
  
// Konstruktoren, getter, setter, toString...  
}
```

Das `uniqueConstraint`-Attribut der `@Table`-Annotation kann genutzt werden, wenn aufgrund der Java-Klasse mittels eines geeigneten Werkzeugs der `CREATE TABLE` generiert werden soll (z.B. mittels des `schemaexport`-Tools von Hibernate). Da die `create.sql`-Dateien hier aber per Hand codiert werden, ist die `@Table`-Annotation natürlich nicht erforderlich. Zur Laufzeit wird diese Annotation ohnehin nicht ausgewertet.

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        manager.persist(new Book("1111", "Pascal", 10));  
        manager.persist(new Book("2222", "Modula", 20));  
    });  
}
```

```
static void demoFind(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final Book b1 = QueryUtil.findByBusinessKey(  
            manager, Book.class, "isbn", "1111");  
        System.out.println(b1);  
        final Book b2 = QueryUtil.findByBusinessKey(  
            manager, Book.class, "isbn", "2222");  
        System.out.println(b2);  
        final Book b3 = QueryUtil.findByBusinessKey(  
            manager, Book.class, "isbn", "3333");  
        System.out.println(b3);  
    });  
}
```

Die Ausgaben:

```
Hibernate: select ... from Book where isbn=?  
Book [1, 1111, 10.0, Pascal]  
Hibernate: select ... from Book where isbn=?  
Book [2, 2222, 20.0, Modula]  
Hibernate: select ... from Book where isbn=?  
null
```

An `findByBusinessKey` werden folgende Parameter übergeben:

- der `PersistenceManager`
- die Klasse des zu erzeugenden Objekts
- die Property / das Attribut, die / das den Business-Key implementiert
- der Business-Key selbst

Und hier die Klasse `QueryUtil` (im `jpa-util`-Projekt):

QueryUtil

```
package jpa.util;
// ...
public class QueryUtil {

    private static final boolean verbose = false;

    private static final Map<Class<?>, PropertyDescriptor[]> pdsMap =
        new HashMap<Class<?>, PropertyDescriptor[]>();

    private static PropertyDescriptor[] getPds(Class<?> cls)
        throws Exception {
        PropertyDescriptor[] pds = pdsMap.get(cls);
        if (pds == null) {
            pds = Introspector.getBeanInfo(
                cls, Object.class).getPropertyDescriptors();
            pdsMap.put(cls, pds);
        }
        return pds;
    }

    @SuppressWarnings("unchecked")
    public static <T> T findByBusinessKey(EntityManager manager,
        Class<T> cls, String propertyName, Object key) {
        final String ALIAS = "o_";
        Query query;
        StringBuilder buf = new StringBuilder();
        buf.append("select ")
            .append(ALIAS).append(" from ")
            .append(cls.getName())
            .append(" ")
            .append(ALIAS)
            .append(" where ");
        if (key.getClass().getAnnotation(Embeddable.class) == null) {
            buf.append(ALIAS)
                .append('.')
                .append(propertyName)
                .append(" = :")
                .append(propertyName);
            String hql = buf.toString();
            if (verbose)
                System.out.println("==> " + hql);
            query = manager.createQuery(hql);
            query.setParameter(propertyName, key);
        }
        else {
            try {
                PropertyDescriptor[] pds = getPds(key.getClass());
                for (int i = 0; i < pds.length; i++) {
                    PropertyDescriptor pd = pds[i];
                    if (i > 0)
```

```
        buf.append(" and ");
        String name = pd.getName();
        buf.append(ALIAS)
            .append('.')
            .append(propertyName)
            .append('.')
            .append(name).append(" = :")
            .append(name);
    }
    String hql = buf.toString();
    if (verbose)
        System.out.println("==> " + hql);
    query = manager.createQuery(hql);
    for (int i = 0; i < pds.length; i++) {
        PropertyDescriptor pd = pds[i];
        Object value = pd.getReadMethod().invoke(key);
        query.setParameter(pd.getName(), value);
    }
}
catch(Exception e) {
    throw new RuntimeException(e);
}
}
try {
    return (T) query.getSingleResult();
}
catch(Exception e) {
    return null;
}
}
```

(Hinweis: Diese Klasse kann auch dann genutzt werden, wenn der Business-Key zusammengesetzt ist – siehe. das "Spezialitäten"-Kapitel).

3.13 Validation

Hibernate stellt die Referenz-Implementierung des Java-Validation-APIs bereit. Im folgenden zeigen wir anhand eines einfachen Beispiels, wie Validierungs-Regeln in den persistenten Klassen hinterlegt werden können (natürlich in Form von Annotations) und wie Hibernate diese Regeln automatisch beim Persistieren von Objekten anwendet.

Eine einfache Tabellendefinition (die ihrerseits abgesehen vom Primary-Key keinerlei Constraints spezifiziert):

```
CREATE TABLE BOOK (  
    ISBN VARCHAR (20),  
    TITLE VARCHAR (128) NOT NULL,  
    PRICE DOUBLE NOT NULL,  
    PRIMARY KEY (ISBN)  
);
```

In der persistente Klasse werden die Validierungsregeln in Form von Annotations hinterlegt – Annotations aus dem Paket `javax.validation.constraints`:

```
package domain;  
// ...  
import javax.validation.constraints.DecimalMax;  
import javax.validation.constraints.DecimalMin;  
import javax.validation.constraints.NotNull;  
import javax.validation.constraints.Size;  
  
@Entity  
public class Book {  
  
    @Id  
    @NotNull  
    @Size(min = 4, max = 4)  
    private String isbn;  
  
    @Basic  
    @NotNull  
    @Size(min = 2, max = 40)  
    private String title;  
  
    @Basic  
    @DecimalMin("1.00")  
    @DecimalMax("99.99")  
    private double price;  
  
    // Konstruoren, getter, setter, toString ...  
}
```

Die `persistence.xml` kann um einen `<validation-mode>`-Eintrag erweitert werden (fehlt dieser Eintrag, so findet eine automatische Validierung statt):

```
<persistence ...>
  <persistence-unit name="library">

    ...

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

    <!-- AUTO, NONE, CALLBACK -->
  </persistence-unit>
</persistence>
```

Hier einige `demo`-Methoden.

Die erste `demo`-Methode persistiert ein Book, das valide ist:

```
static void demoOkay(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
    });
}
```

In allen folgenden `demo`-Methoden wird eine Exception geworfen – eine Exception des Typs `javax.validation.ConstraintViolationException`:

```
static void demoIllegalIsbn(TransactionTemplate tt) {
    try {
        tt.run(manager -> {
            manager.persist(new Book("22", "Modula", 20));
        });
    }
    catch (Exception e) {
        System.out.println("Expected: " + e + e.getCause());
    }
}
```

```
Expected: javax.persistence.RollbackException:
Error while committing the transaction
javax.validation.ConstraintViolationException:
List of constraint violations:[
    ConstraintViolationImpl{
        interpolatedMessage='muss zwischen 4 und 4 liegen',
        propertyPath=isbn,
        rootBeanClass=class domain.Book,
        messageTemplate=
            '{javax.validation.constraints.Size.message}'
    }
]
```

```
static void demoIllegalTitle(TransactionTemplate tt) {
    try {
        tt.run(manager -> {
            manager.persist(new Book("3333", null, 30));
        });
    }
}
```

```
    });  
    }  
    catch (Exception e) {  
        System.out.println("Expected: " + e + e.getCause());  
    }  
}
```

Expected: javax.persistence.RollbackException: ...

```
static void demoIllegalPrice(TransactionTemplate tt) {  
    try {  
        tt.run(manager -> {  
            manager.persist(new Book("4444", null, 90000));  
        });  
    }  
    catch (Exception e) {  
        System.out.println("Expected: " + e.getCause());  
    }  
}
```

Expected: javax.persistence.RollbackException: ...

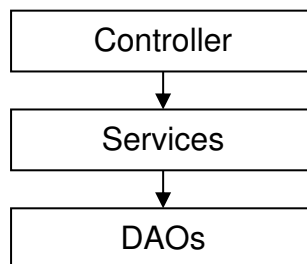
Nach Aufruf aller vier `demo`-Methoden ist nur eine einzige Transaktion committed worden:

```
BOOK  
ISBN TITLE  PRICE  
-----  
1111 Pascal  10.0  
-----
```

4 Ein kleines Service-Framework

Während die bislang benutzte Klasse `TransactionTemplate` die Implementierung von kleinen Beispielprogrammen wesentlich vereinfacht (und daher auch im folgenden immer wieder genutzt werden wird), ist sie für "richtige" Anwendungen nur bedingt geeignet.

Eine "richtige" Anwendung besteht natürlich nicht nur aus der `Application`-Klasse. Sie wird Klassen besitzen, in denen die Fachlogik und die Persistenzlogik gekapselt sind. Eine solche Anwendung könnte etwa folgende Schichten besitzen:



Die DAO-Schicht (Data Access Objects) enthält Klassen, welche die JPA-Zugriffe enthalten. Nur in dieser Schicht sollte die verwendete Persistenz-Technologie bekannt sein (z.B. JPA, oder Hibernate pur, oder JDBC...). Die Service-Schicht implementiert die Fachlogik (unter Zuhilfenahme der DAO-Schicht). Und der Controller stößt aufgrund von Benutzereingaben die jeweils erforderliche Fachlogik an.

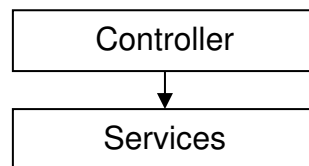
Dann stellt sich die Frage, wo die Transaktionskontrolle angesiedelt sein muss. Im folgenden wird vorausgesetzt, dass jede öffentliche Service-Methode mit genau einer einzigen Transaktion verbunden ist. Für den Aufruf jeder Service-Methode muss also eine Transaktion gestartet werden. (Sollen mehrere Service-Methoden in einer einzigen Transaktion laufen, benötigt man dann eine weitere Service-Methode, welche diese Methoden klammert.)

Die Service-Methoden selbst sollten aber für die Transaktionssteuerung nicht zuständig sein (sie sollten nur Fachlogik beinhalten). Also muss diese Steuerung oberhalb der Service-Schicht angesiedelt sein. Der Controller könnte nun jeden Aufruf einer Service-Methode in einer `Function` oder einem `Consumer` implementieren, welche (welcher) dann von einem `TransactionTemplate` aufgerufen wird. Der Controller (z.B. die `main`-Methode einer Anwendung) könnte dann etwa folgenden Code beinhalten:

```
TransactionTemplate tt = new TransactionTemplate(factory);
BookService bookService = new BookService();
tt.run(manager -> bookService.insertBook(manager, new Book(...)));
```


In jedem Lambda-Ausdruck würde somit genau eine einzige Service-Methode aufgerufen. Jeder Service-Methode müsste neben den fachlichen Parametern noch der `EntityManager` als Parameter übergeben werden. Die Service-Methode benötigt den `EntityManager` natürlich nicht selbst – sie müsste ihn aber an den Aufruf der Methoden der DAO-Schicht weiterreichen. Eine solche Lösung ist natürlich nicht unbedingt erstrebenswert...

Im folgenden wird eine Lösung vorgestellt, welche auf dem `ThreadLocal`-Konzept beruht. Der Einfachheit halber wird davon ausgegangen, dass die JPA-Zugriffe direkt in der Service-Schicht stattfinden (dass also keine eigene DAO-Schicht existiert).



Die im folgenden vorgestellte Lösung wäre allerdings auch dann tragfähig, wenn Fachlogik (Service-Schicht) und Persistenzlogik (DAO-Schicht) getrennt wären.

Wir beginnen damit, den benötigten Service in einem Interface zu spezifizieren (Services und auch DAOs sollten immer zunächst mittels eines Interfaces spezifiziert werden – einer der Gründe hierfür wird im nächsten Abschnitt deutlich werden):

BookService

```
package services;
// ...
public interface BookService {
    public abstract void insertBooks(Book... books);
    public abstract Book findBook(String isbn);
    public abstract void deleteBook(String isbn);
    public abstract void updateBook(String isbn, String title);
    public abstract List<Book> findBooks();
}
```

Man beachte, dass den Methoden ausschließlich fachliche Parameter übergeben werden. Würde diesen Methoden noch zusätzlich ein `EntityManager` übergeben, wäre das Interface "verunreinigt" – man würde dem (fachlichen) Interface bereits die verwendete (technische) Persistenz-Technologie ansehen.

4.1 Verwendung von ThreadLocal

Wir unterstellen nun die Existenz eine "Framework"-Klasse, mittels derer die Implementierung dieses Interface den "aktuellen" `EntityManager` ermitteln kann (wie diese Klasse implementiert ist, wird später erläutert):

```
package jpa.util;
// ...
public class EntityManagerThreadLocal {

    public static EntityManager getCurrentEntityManager() { ... }
    // ...
}
```

Dann kann das obige Interface wie folgt implementiert werden:

BookServiceImpl

```
package services;
// ...
public class BookServiceImpl implements BookService {

    private EntityManager getManager() {
        return EntityManagerThreadLocal.getCurrentEntityManager();
    }

    public void insertBooks(Book... books) {
        for (final Book book : books)
            this.getManager().persist(book);
    }

    public Book findBook(String isbn) {
        return this.getManager().find(Book.class, isbn);
    }

    public void deleteBook(String isbn) {
        final Book book = this.getManager().find(Book.class, isbn);
        this.getManager().remove(book);
    }

    public void updateBook(String isbn, String title) {
        final Book book = this.getManager().find(Book.class, isbn);
        book.setTitle(title);
    }

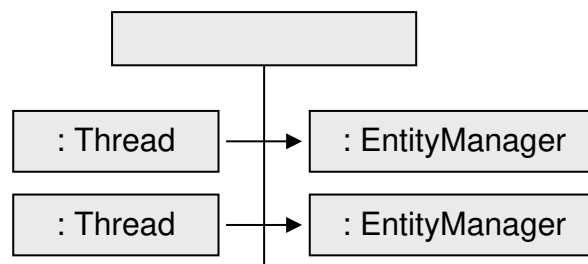
    public List<Book> findBooks() {
        final TypedQuery<Book> query = this.getManager().createQuery(
            "select b from Book b", Book.class);
        return query.getResultList();
    }
}
```

Nur die Implementierung ist somit von der Klasse `EntityManager` abhängig.

(Bei der Verwendung von DAOs würden die Service-Methoden DAO-Methoden aufrufen, welche ebenfalls nur fachliche Parameter besitzen. Und nur in diesen DAO-Methoden würde dann der `EntityManager` ermittelt werden müssen. Die Service-Methoden müssten den `EntityManager` überhaupt nicht kennen...)

Wie kann die oben verwendete Klasse `EntityManagerThreadLocal` implementiert werden – genauer: wie kann deren statische Methode `getCurrentEntityManager` implementiert werden?

Einige Vorüberlegungen: Die bisherigen Beispiele waren allesamt single-threaded. In einer "realistischen" Anwendung (insbesondere in einer Web-basierten Anwendung) muss aber stets mit mehreren Threads gerechnet werden. Eine realistische Anwendung ist multi-threaded. Zwar benötigt man auch in einer solchen Anwendung nur eine einzige `EntityManagerFactory` – aber für jeden Thread einen eigenen `EntityManager`. Wollen also z.B. zwei Threads "gleichzeitig" auf einen Service zugreifen, benötigt jeder dieser Threads einen eigenen `EntityManager`. Nachdem also ein `EntityManager` in einem Thread erzeugt wurde, muss dieser Manager an diesen Thread "gebunden" werden. Und die Methode `getCurrentEntityManager` muss dann jeweils den mit dem jeweiligen Thread verbundenen Manager zurückliefern. Man benötigt also eine Abbildung von Thread-Objekten auf `EntityManager`-Objekte (und diese Abbildung muss "global" sein – also `static`):



Zu diesem Zweck kann die Java-Standardklasse `ThreadLocal` genutzt werden. Hier die wesentlichen Methoden dieser Klasse:

```
package java.lang;

public class ThreadLocal<T> {
    public void set(T obj) { ... }
    public T get() { ... }
    public void remove() { ... }
}
```

Die `set`-Methode dieser Klasse kann benutzt werden, um eine Zuordnung des aktuellen Threads zu irgendeinem Objekt (z.B. zu einem `EntityManager`) in ein `ThreadLocal`-

Objekt einzutragen. Mittels der `get`-Methode kann das mit dem aktuellen `Thread` assoziierte Objekt ermittelt werden. Und mittels `remove` kann der dem aktuellen `Thread` entsprechende Eintrag aus dem `ThreadLocal` entfernt werden. (Man beachte, dass der aktuelle `Thread` nirgendwo als Parameter übergeben wird – dieser kann in den Methoden der Klasse mittels des Aufrufs von `Thread.currentThread()` ermittelt werden).

Hier nun die "Framework"-Klasse `EntityManagerThreadLocal`:

EntityManagerThreadLocal

```
package jpa.util;
// ...
public class EntityManagerThreadLocal {

    private final static ThreadLocal<EntityManager> managers =
        new ThreadLocal<EntityManager>();

    public static EntityManager getCurrentEntityManager() {
        EntityManager manager = managers.get();
        if (manager == null)
            throw new RuntimeException("missing currentEntityManager");
        return manager;
    }
    public static void setEntityManager(EntityManager manager) {
        if (managers.get() != null)
            throw new RuntimeException(
                "EntityManager already bound to current thread");
        managers.set(manager);
    }
    public static void removeEntityManager() {
        EntityManager manager = managers.get();
        if (manager == null)
            throw new RuntimeException(
                "no EntityManager bound to current thread");
        managers.remove();
    }
    public static boolean isEntityManagerBound() {
        return managers.get() != null;
    }
}
```

`EntityManagerThreadLocal` definiert ausschließlich `static`-Elemente. `managers` zeigt auf ein `ThreadLocal`-Objekt, welches die Zuordnung von `Threads` zu `EntityManagern` gestattet. `getCurrentEntityManager` liefert den mit dem aktuellen `Thread` assoziierten `EntityManager` zurück. Mittels `setEntityManager` kann ein `EntityManager` mit dem aktuellen `Thread` assoziiert werden. `removeEntityManager` entfernt den für den aktuellen `Thread` gespeicherten Eintrag. Und mittels `isEntityManagerBound` kann ermittelt werden, ob der aktuelle `Thread` bereits mit einem `EntityManager` verbunden ist. Man beachte jeweils die Absicherungen in den Methoden.

Mittels des `ThreadLocal`-Konzepts kann also ein `EntityManager` transparent an aufgerufene Methoden weitergereicht werden (die ihn ebenso transparent an weitere Methoden weiterleiten können). `EntityManager` müssen somit also nicht mehr als explizite Parameter übergeben werden.

Hinweis: Bei der Verwendung von `ThreadLocal` ist unbedingt zu beachten, dass diejenige Instanz, die ein Objekt an einen `ThreadLocal` bindet, die entsprechende Bindung auch wieder aus dem `ThreadLocal` entfernt. Ansonsten kann die Verwendung von `ThreadLocal` – insbesondere dann, wenn Thread-Pooling verwendet wird – zu ernsthaften Problemen führen!

Eine einfache Transaktion könnte nun wie folgt implementiert werden:

Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();
        final EntityManagerFactory factory =
            Configuration.getEntityManagerFactory();
        try {
            demo(factory);
        }
        finally {
            factory.close();
        }
    }

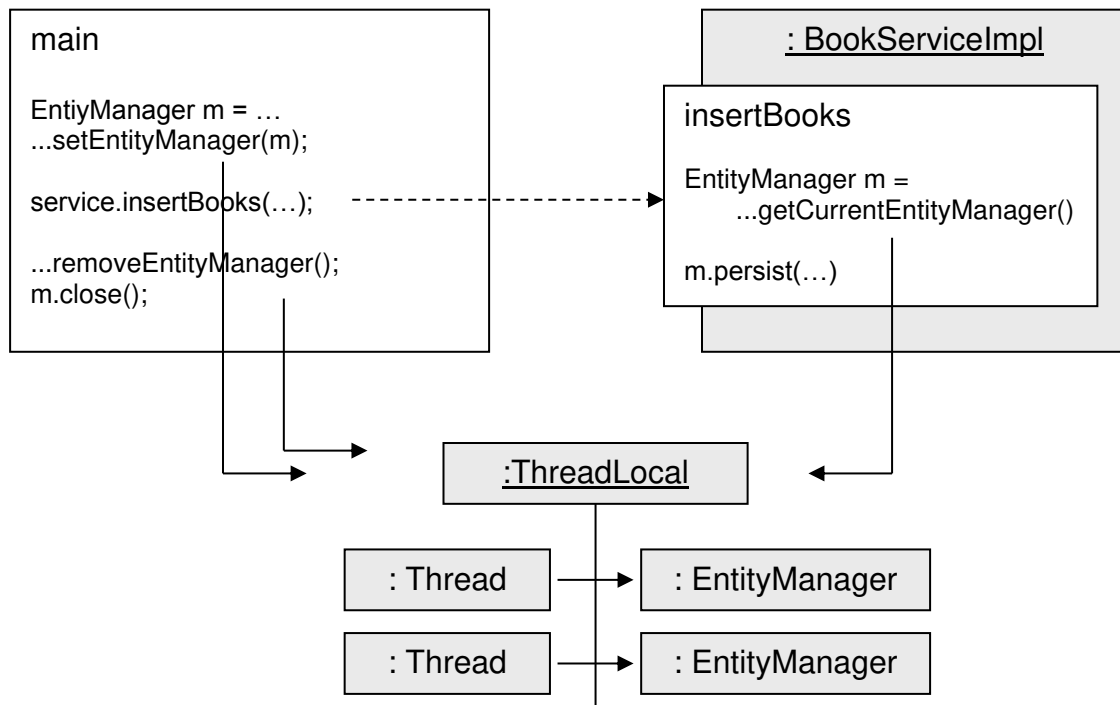
    static void demo(EntityManagerFactory factory) {
        Util.mlog();
        final BookService service = new BookServiceImpl();
        final EntityManager manager = factory.createEntityManager();
        EntityManagerThreadLocal.setEntityManager(manager);
        final EntityTransaction transaction = manager.getTransaction();
        try {
            transaction.begin();
            service.insertBooks(
                new Book("1111", "Pascal", 10),
                new Book("2222", "Modula", 20),
                new Book("3333", "Oberon", 30)
            );
            transaction.commit();
        }
        catch (final RuntimeException e) {
            System.out.println(e);
            if (transaction.isActive())
                transaction.rollback();
            throw e;
        }
    }
}
```

```

    finally {
        EntityManagerThreadLocal.removeEntityManager();
        manager.close();
    }
}

```

Hier noch einmal ein Bild, welches die Funktionsweise der obigen Anwendung verdeutlicht:



Natürlich könnte man nun die `TransactionTemplate`-Klasse erweitern – die `EntityManagerThreadLocal`-Aufrufe `setEntityManager` und `removeEntityManager` könnten in die `run`-Methode der Klasse integriert werden. Zudem könnten die `accept`- bzw. `apply`-Methoden der `Consumer` resp. der `Functions` parameterlos sein – ihnen müsste kein `EntityManager` mehr übergeben werden. Dieser Weg soll hier aber nicht besprochen werden...

Stattdessen soll im folgenden eine andere Möglichkeit vorgestellt werden, die Transaktionssteuerung zu abstrahieren. Diese Variante setzt aber voraus, dass der auszuführende Code tatsächlich in Service-Klassen (mit oder ohne DAOs) implementiert ist. Sie setzt weiterhin voraus, dass die Services (resp. DAOs) über Interfaces spezifiziert sind.

4.2 DelegatingEntityManager

Man betrachte noch einmal die bislang verwendete Klasse `BookServiceImpl`:

```
package services;
// ...
public class BookServiceImpl implements BookService {

    private EntityManager getManager() {
        return EntityManagerThreadLocal.getCurrentEntityManager();
    }

    public void insertBooks(Book... books) { ... }
    // ...
}
```

Jede Methode dieser Klasse benötigt den mit dem aktuellen Thread assoziierten `EntityManager`. Dieser wird mittels der `EntityManagerThreadLocal`-Methode `getCurrentEntityManager()` ermittelt. Das funktioniert alles problemlos. Allerdings ist nun die Implementierungsklasse abhängig von einer "zufälligen" Klasse unseres "zufälligen" Mini-Frameworks: `EntityManagerThreadLocal`. Es wäre schöner, wenn die Service-Klasse ohne Verwendung von `EntityManagerThreadLocal` implementiert werden könnte.

Das Mini-Framework muss zu diesem Zweck nur um einige Kleinigkeiten erweitert werden. Hier aber zunächst das Resultat – die neue Variante von `BookServiceImpl`:

BookServiceImpl

```
package services;
// ...
public class BookServiceImpl implements BookService {

    private final EntityManager manager;

    public BookServiceImpl(EntityManager manager) {
        this.manager = manager;
    }

    public void insertBooks(Book... books) {
        for (final Book book : books)
            this.manager.persist(book);
    }

    public Book findBook(String isbn) {
        return this.manager.find(Book.class, isbn);
    }

    // ...
}
```

Der `EntityManager` wird nun offenbar über den Aufruf des Konstruktors von "außen" gesetzt. Er muss dem `BookServiceImpl`-Objekt "injiziert" werden ("Dependency Injection"). Der injizierte Manager wird der Instanzvariablen `manager` zugewiesen. Und die eigentlichen Service-Methoden beziehen sich dann über diese Instanzvariable auf den `EntityManager`.

Man beachte, dass das Interface `BookService` nicht(!) verändert wurde.

Das Problem besteht nun darin: es gibt ein einziges `BookServiceImpl`-Objekt. Auf dieses Objekt greifen der Reihe nach oder aber auch gleichzeitig mehrere Threads zu. Und jeder Thread benötigt seinen eigenen `EntityManager`. Daraus folgt: der `EntityManager`, der dem `BookServiceImpl`-Objekt injiziert wird, kann nicht der "eigentliche" `EntityManager` sein. Zum Glück ist `EntityManager` nur ein Interface...

Es muss sich also um ein Objekt einer Klasse handeln, welche ebenfalls das Interface `EntityManager` implementiert – aber derart, dass alle Methodenaufrufe delegiert werden an die entsprechenden Methoden des mit dem aktuellen Thread assoziierten "realen" `EntityManagers`. Es muss sich um ein Objekt handeln, dass auf den jeweils "realen" Manager zugreift:

DelegatingEntityManager

```
package jpa.util;
// ...
public class DelegatingEntityManager implements EntityManager {

    // Singleton
    private static final DelegatingEntityManager instance =
        new DelegatingEntityManager();

    public static DelegatingEntityManager getInstance() {
        return instance;
    }

    private DelegatingEntityManager() {
    }

    @Override
    public void persist(Object obj) {
        EntityManagerThreadLocal.getCurrentEntityManager()
            .persist(obj);
    }

    @Override
    public void remove(Object obj) {
        EntityManagerThreadLocal.getCurrentEntityManager()
            .remove(obj);
    }

    @Override
```



```

public <T> T find(Class<T> cls, Object key) {
    return EntityManagerThreadLocal.getCurrentEntityManager()
        .find(cls, key);
}
@Override
public <T> TypedQuery<T> createQuery(String jpql, Class<T> cls) {
    return EntityManagerThreadLocal.getCurrentEntityManager()
        .createQuery(jpql, cls);
}
// ...
}

```

Alle Methoden delegieren jeweils an den mit dem aktuellen Thread verbundenen EntityManager. Dieser wird – wie zuvor in der Klasse `BookServiceImpl` – mittels der `EntityManagerThreadLocal`-Methode `getCurrentEntityManager()` ermittelt. Man beachte, dass man mit einem einzigen `DelegatingEntityManager` auskommt (die Klasse ist eine Singleton-Klasse).

Hier die neue Demo-Methode:

```

static void demo(EntityManagerFactory factory) {
    Util.mlog();
    final BookService service =
        new BookServiceImpl(DelegatingEntityManager.getInstance());
    final EntityManager manager =
        factory.createEntityManager();
    EntityManagerThreadLocal.setEntityManager(manager);
    final EntityTransaction transaction = manager.getTransaction();
    try {
        transaction.begin();
        service.insertBooks(
            new Book("1111", "Pascal", 10),
            new Book("2222", "Modula", 20),
            new Book("3333", "Oberon", 30)
        );
        transaction.commit();
    }
    catch (final RuntimeException e) {
        System.out.println(e);
        if (transaction.isActive())
            transaction.rollback();
        throw e;
    }
    finally {
        EntityManagerThreadLocal.removeEntityManager();
        manager.close();
    }
}

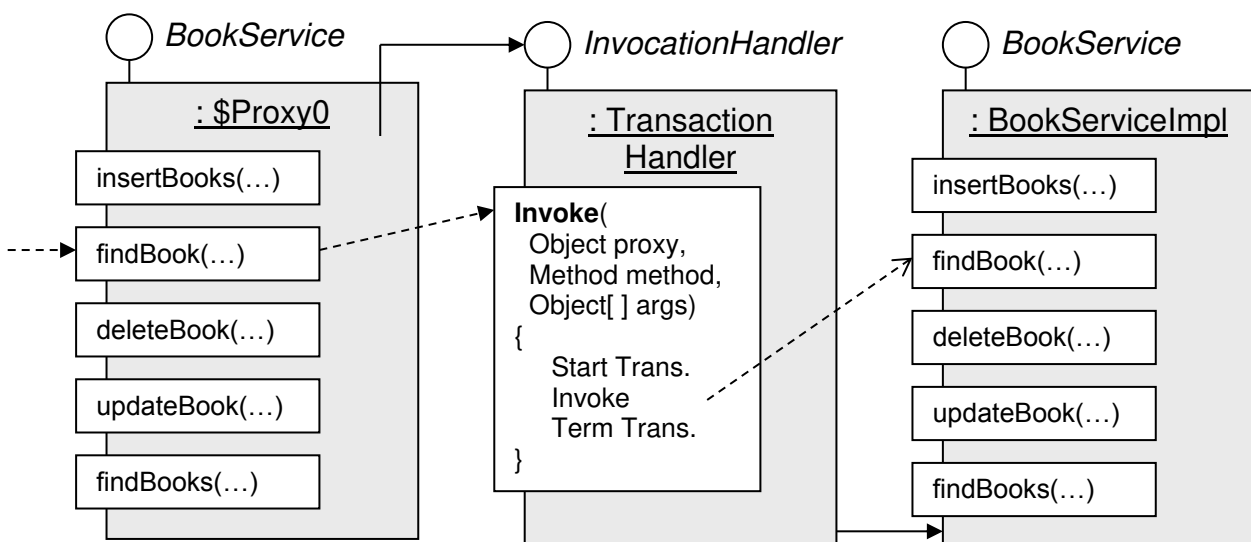
```

4.3 Proxies

Java kennt bekanntlich das Konzept der Dynamic Proxies. Mittels dieses Konzepts können zur Laufzeit Klassen generiert werden, welche ein bestimmtes fachliches Interface implementieren. Bei der Instanziierung einer solchen zur Laufzeit generierten Klasse wird dann ein Handler-Objekt übergeben, dessen Klasse ein technisches Interface implementiert: das Interface `InvocationHandler`. Dieses Interface deklariert nur eine einzige Methode, die Methode `invoke`. Alle Methoden der generierten Proxy-Klasse delegieren an diese `invoke`-Methode. Sie übergeben dieser `invoke`-Methode u.a. ein `Method`-Objekt, welches die entsprechende Interface-Methode beschreibt - und sie reichen ihre eigenen Aufrufparameter als `Object`-Array an die `invoke`-Methode weiter. Besitzt nun dieses Handler-Objekt eine Referenz auf die "reale" Implementierung der fachlichen Interfaces, so kann die `invoke`-Methode des Handler-Objekts an die entsprechende "reale" Methode delegieren. Vorher aber kann sie beliebige vorbereitende Aktionen ausführen. Und nach Rückkehr der "realen" Methode kann sie ebenso beliebige Abschlussarbeiten ausführen.

Die `invoke`-Methode eines solchen `InvocationHandlers` ist somit der richtige Kandidat für die Transaktionssteuerung. Sie wird ungefähr genau diejenigen Aktionen ausführen, welche auch von der `run`-Methode der `TransactionTemplate`-Klasse ausgeführt werden. Zusätzlich aber wird sie den von ihr erzeugten `EntityManager` in den von `EntityManagerThreadLocal` verwalteten `ThreadLocal` registrieren und wieder deregistrieren. Die Handler-Klasse wird als `TransactionHandler` bezeichnet werden. (Hinweis: EJB-Container benutzen dasselbe hier vorgestellte Konzept...)

Zunächst ein Schaubild:



`$Proxy0` sei der Name der zur Laufzeit generierten Proxy-Klasse. Sie implementiert z.B. das Interface `BookService`. Das Proxy-Objekt besitzt eine Referenz auf einen `TransactionHandler`. Diese Klasse implementiert das Interface `InvocationHandler`. Der `TransactionHandler` hat eine Referenz auf das `BookServiceImpl`-Objekt – kennt dieses Objekt aber nur abstrakt als `Object` (die entsprechende Referenz ist vom Typ `Object`).

Die `invoke`-Methode des `TransactionHandlers` wird die Transaktion starten. Dann benutzt sie das ihr vom Proxy übergebene `Method`-Objekt, um die "reale" Methode auf den `BookServiceImpl` aufzurufen. Nach Rückkehr dieser Methode kann dann in `invoke` die Transaktion beendet werden.

Hier zunächst eine Anwendung, welche den neuen `TransactionHandler` nutzt (damit man sieht, dass dieses Konzept offensichtlich äußerst hilfreich ist):

Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) {
        Db.aroundAppl();
        final EntityManagerFactory factory =
            Configuration.getEntityManagerFactory();
        try {
            final BookService service =
                TransactionHandler.createProxy(
                    factory, BookService.class, new BookServiceImpl(
                        DelegatingEntityManager.getInstance()));
            demo(service);
        }
        finally {
            factory.close();
        }
    }

    static void demo(BookService service) {
        Util.mlog();
        service.insertBooks(
            new Book("1111", "Pascal", 10),
            new Book("2222", "Modula", 20),
            new Book("3333", "Oberon", 30)
        );

        final Book book = service.findBook("2222");
        System.out.println(book);

        service.deleteBook("1111");

        service.updateBook("2222", "Modula-2");
    }
}
```

```

        final List<Book> bookList = service.findBooks();
        bookList.forEach(System.out::println);
    }
}

```

An die `createProxy`-Methode der Klasse `TransactionHandler` wird das vom Proxy zu implementierende Interface (hier: `BookService.class`) und ein Objekt einer Implementierungsklasse (hier: ein `BookServiceImpl`-Objekt) übergeben. Der Aufruf dieser `createProxy`-Methode resultiert in genau derjenigen Konstellation, welche im obigen Schaubild dargestellt ist. `service` zeigt also auf das Proxy-Objekt, an welchem ein `TransactionHandler` hängt, der seinerseits dann auf das `BookServiceImpl`-Objekt verweist.

Jeder Aufruf einer der `BookService`-Methoden läuft nun transparent in einer eigenen Transaktion, deren `EntityManager` im `ThreadLocal` von `EntityManagerThreadLocal` registriert ist. Die Methoden von `BookServiceImpl` können den benötigten `EntityManager` also wie im letzten Abschnitt per `getCurrentEntityManager` ermitteln.

Hier die Ausgaben der obigen Beispiel-Anwendung:

```

+-----+
| demo  |
+-----+
-----
insertBooks
-----
Hibernate: insert into Book (...) values (?, ?, ?)
Hibernate: insert into Book (...) values (?, ?, ?)
Hibernate: insert into Book (...) values (?, ?, ?)
-----
findBook
-----
Hibernate: select ... from Book where isbn=?
Book [2222, 20.0, Modula]
-----
deleteBook
-----
Hibernate: select ... from Book where isbn=?
Hibernate: delete from Book where isbn=?
-----
updateBook
-----
Hibernate: select ... from Book where isbn=?
Hibernate: update Book set price=?, title=? where isbn=?
-----
findBooks
-----
Hibernate: select ... from Book
Book [2222, 20.0, Modula-2]

```

Book [3333, 30.0, Oberon]

Hier schließlich die Klasse `TransactionHandler`, deren genaues Studium dem Leser überlassen bleiben soll:

TransactionHandler

```
package jpa.util;
// ...
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class TransactionHandler implements InvocationHandler {

    @SuppressWarnings("unchecked")
    public static <T> T createProxy(EntityManagerFactory factory,
        Class<T> iface, Object realObject) {
        return (T) Proxy.newProxyInstance(
            Thread.currentThread().getContextClassLoader(),
            new Class[] { iface },
            new TransactionHandler(factory, realObject));
    }

    private final EntityManagerFactory factory;
    private final Object target;

    public TransactionHandler(EntityManagerFactory factory, Object target) {
        this.factory = factory;
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        if (EntityManagerThreadLocal.isEntityManagerBound()) {
            try {
                return method.invoke(this.target, args);
            }
            catch (final Throwable t) {
                if (t instanceof InvocationTargetException)
                    throw ((InvocationTargetException)t).getCause();
                throw t;
            }
        }

        printHeader(method.getName());

        final EntityManager manager =
            this.factory.createEntityManager();
        EntityManagerThreadLocal.setEntityManager(manager);
        final EntityManagerTransaction transaction = manager.getTransaction();
        try {
```

```
        transaction.begin();
        final Object result = method.invoke(this.target, args);
        transaction.commit();
        return result;
    }
    catch (final Throwable t) {
        if (transaction.isActive())
            transaction.rollback();
        if (t instanceof InvocationTargetException)
            throw ((InvocationTargetException)t).getCause();
        throw t;
    }
    finally {
        manager.close();
        EntityManagerThreadLocal.removeEntityManager();
    }
}

private static void printHeader(String name) {
    final String line = "-----";
    System.out.println(line);
    System.out.println(name);
    System.out.println(line);
}
}
```

Die Klasse ist nur abhängig von `javax.persistence` und `java.lang.reflect` – von den konkreten Services ist sie unabhängig. Sie kann also für alle Services genutzt werden.

Hinweise:

Diese Klasse kann selbstverständlich auch in einer realen Web-Anwendung (ohne EJB) eingesetzt werden.

In einem EJB-Container übernimmt dieser genau diejenigen Aufgaben, welche die hier vorgestellte Klasse erledigt. Sie wird in einem solchen Kontext also nicht benötigt.

4.4 Multithreading

Im folgenden wird gezeigt, dass die oben entwickelte Lösung auch in einer multi-threaded Umgebung funktioniert – denn für einen solchen Kontext ist die Lösung ja entwickelt worden.

Zunächst einige kleinere Änderungen an der Klasse `BookServiceImpl`:

BookServiceImpl

```
package services;
// ...
public class BookServiceImpl implements BookService {

    private final EntityManager manager;

    public BookServiceImpl(EntityManager manager) {
        this.manager = manager;
    }

    @Override
    public void insertBooks(Book... books) {
        for (final Book book : books) {
            Util.tlog("before persist " +
                this.manager.getTransaction());
            this.manager.persist(book);
            try {
                Thread.sleep(2000);
            }
            catch (final InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
    // ...
}
```

Die `insertBooks`-Methode wird derart erweitert, dass in jedem Schleifendurchlauf zunächst der aktuelle Thread und die aktuelle Transaktion ausgegeben wird. Nach jedem Einfügen eines `Book`-Objekts legt die Methode sich dann 2 Sekunden schlafen.

In der folgenden Applikation werden zwei Threads parallel ausgeführt. Jeder der beiden Threads ruft `insertBooks` mit jeweils drei `Book`-Objekten auf. Der zweite Thread wird eine Sekunde später gestartet als der erste (mit dem Ergebnis, dass sich die `persist`-Aufrufe von `BookServiceImpl` bezüglich der beiden Threads "verzahnen" werden):

Application

```
package appl;
```

```
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        Db.aroundAppl();
        final EntityManagerFactory factory =
            Configuration.getEntityManagerFactory();
        try {
            final BookService service =
                TransactionHandler.createProxy(
                    factory, BookService.class, new BookServiceImpl(
                        DelegatingEntityManager.getInstance()));
            demo(service);
        }
        finally {
            factory.close();
        }
    }

    static void demo(BookService service) throws Exception {
        Util.mlog();
        final Thread t1 = new Thread(() ->
            service.insertBooks(
                new Book("1111", "Pascal", 10),
                new Book("2222", "Modula", 20),
                new Book("3333", "Oberon", 30))
        );
        final Thread t2 = new Thread(() ->
            service.insertBooks(
                new Book("4444", "C", 40),
                new Book("5555", "C++", 50),
                new Book("6666", "C#", 60))
        );
        t1.start();
        Thread.sleep(1000);
        t2.start();
        t1.join();
        t2.join();
    }
}
```

Hier die Ausgaben (sql-show ist ausgeschaltet):

```
+-----+
| demo  |
+-----+
insertBooks
-----
[13] before persist ...TransactionImpl@737971d1
-----
insertBooks
-----
[14] before persist ...TransactionImpl@238b84e8
```



```
[13] before persist ...TransactionImpl@737971d1  
[14] before persist ...TransactionImpl@238b84e8  
[13] before persist ...TransactionImpl@737971d1  
[14] before persist ...TransactionImpl@238b84e8
```

Man erkennt, dass die beiden Threads jeweils eine eigene Transaktion nutzen.

5 Spezialitäten

Dieses Kapitel behandelt einige "Spezialitäten" von JPA. (Falls die Zeit knapp ist, kann dieses Kapitels übersprungen werden...)

Es geht um folgende Features:

- Zusammengesetzte Schlüssel als Primary Keys
- Sog. Embeddables
- Sog. ElementCollections
- Sog. Secondary Tables
- XML-Mapping als Alternative / Ergänzung zu Annotations

5.1 Business-Keys: Composite Keys

Sofern Business-Keys als Primary-Keys verwendet werden, kann es natürlich leicht passieren, dass solche Schlüssel mehrere Komponenten besitzen: dass es sich um zusammengesetzte Schlüssel handelt. Ein `AUTHOR` z.B. sei identifiziert durch die Kombination von `LASTNAME` und `FIRSTNAME`:

create.sql

```
create table AUTHOR (  
    LASTNAME varchar (64),  
    FIRSTNAME varchar (64),  
    TEXT varchar (128),  
    primary key (LASTNAME, FIRSTNAME)  
)
```

Um solche zusammengesetzten Schlüssel abzubilden, bedarf es einer eigenen, zusätzlichen Klasse, welche den Aufbau solcher Schlüssel beschreibt:

AuthorKey

```
package domain;  
  
public class AuthorKey {  
  
    private String lastname;  
    private String firstname;  
  
    // Konstruktoren, getter, setter, toString...  
  
    @Override  
    public boolean equals(Object obj) { ... }  
  
    @Override  
    public int hashCode() { ... }  
}
```

Die Key-Klasse darf NICHT als `@Entity`-Klasse definiert werden!

Die Key-Klasse benötigt auf jeden Fall einen parameterlosen Konstruktor.

Und es sollten die `Object`-Methoden `equals` und `hashCode` überschrieben werden - und zwar derart, dass die Implementierung beider Methoden alle Schlüsselkomponenten in die Berechnung einbezieht.

Author

```
package domain;
```

```
// ...
@Entity
@IdClass(AuthorKey.class)
public class Author {

    @Id
    private String lastname;

    @Id
    private String firstname;

    @Basic
    private String text;

    // Konstruktoren, getter, setter, toString...
}
```

Neben der `@Entity`-Annotation besitzt die Klasse eine `@IdClass`-Annotation, welche als Attribut das Class-Objekt der Key-Klasse besitzt:

```
@Entity
@IdClass(AuthorKey.class)
```

Zudem muss die Klasse Attribute besitzen, die den Attributen der Key-Klasse äquivalent sind: `lastname`, `firstname`. Diese Attribute müssen mit `@Id` gekennzeichnet sein.

Hier eine Anwendung der beiden obigen Klassen:

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Author(
            "Wirth", "Niklaus", "Father of Pascal"));
        manager.persist(new Author(
            "Meyer", "Bertrand", "Father of Eiffel"));
    });
}
```

```
static void demoFind(TransactionTemplate tt) {
    tt.run(manager -> {
        final Author wirth = manager.find(
            Author.class, new AuthorKey("Wirth", "Niklaus"));
        System.out.println(wirth);
        final Author meyer = manager.find(
            Author.class, new AuthorKey("Meyer", "Bertrand"));
        System.out.println(meyer);
    });
}
```

Hier die Ausgaben:

```
Author [Niklaus, Wirth, Father of Pascal]  
Author [Bertrand, Meyer, Father of Eiffel]
```

Der `find`-Methode wird neben dem `Class`-Objekt, welches die Klasse der zu erzeugenden Objekte bezeichnet, zusätzlich der Primary Key übergeben - in Gestalt eines Objekts der entsprechenden Key-Klasse. Z.B.:

```
final Author wirth = manager.find(  
    Author.class, new AuthorKey("Wirth", "Niklaus"));
```

5.2 Business-Keys: Embedded Composite Keys

Im folgenden wird eine Variante der obigen Lösung vorgestellt.

Die Variante beruht auf derselben Tabellendefinition wie die letzte Lösung.

AuthorKey

```
package domain;
// ...
@Embeddable
public class AuthorKey {

    private String lastname;
    private String firstname;

    AuthorKey() { }

    public AuthorKey(String lastname, String firstname) { ... }

    public String getLastName() { ... }
    void setLastName(String lastname) { ... }

    public String getFirstname() { ... }
    void setFirstname(String firstname) { ... }

    @Override
    public String toString() { ... }

    @Override
    public boolean equals(Object obj) { ... }

    @Override
    public int hashCode() { ... }
}
```

Der einzige Unterschied zur AuthorKey-Klasse im letzten Abschnitt besteht in der Kennzeichnung der Klasse als @Embeddable.

Author

```
package domain;
// ...
@Entity
public class Author {

    private AuthorKey key;
    private String text;

    Author() { }
```

```
public Author(String lastname, String firstname, String text) {
    this.key = new AuthorKey(lastname, firstname);
    this.text = text;
}

public Author(String lastname, String firstname) {
    this(lastname, firstname, null);
}

@EmbeddedId
public AuthorKey getKey() { ... }
public void setKey(AuthorKey key) { ... }

public String getText() { ... }
public void setText(String text) { ... }

@Override
public String toString() { ... }
}
```

Die Klasse ist nur als `@Entity` gekennzeichnet (`@IdClass` entfällt hier). Stattdessen ist statt der bisherigen beiden Attribute `lastname` und `firstname` ein einziges Attribut vom Typ `AuthorKey` definiert: `key`. Diese Property ist als `@EmbeddedId` gekennzeichnet.

Natürlich hätte man auch noch weitere (als `@Transient` gekennzeichnete!) Attribute für den direkten Zugriff auf Nach- und Vornamen einbauen können.

Als Testapplikation wird dieselbe Klasse wie im letzten Abschnitt verwendet. Und somit sind auch die Ausgaben identisch mit denjenigen im letzten Abschnitt.

Wahrscheinlich ist die hier vorstellte zweite Variante der ersten Varianten vorzuziehen.

5.3 FindByBusinessKey

Die bereits im letzten Kapitel vorgestellte Hilfsmethode `findByBusinessKey` funktioniert auch bei zusammengesetzten Schlüsseln.

Die Datenbank-Definition, die Klassen `AuthorKey` und `Author` sind unverändert aus dem letzten Projekt übernommen worden.

Hier die neue Anwendung:

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(
            new Author("Wirth", "Niklaus", "Father of Pascal"));
        manager.persist(
            new Author("Meyer", "Bertrand", "Father of Eiffel"));
    });
}
```

```
static void demoFind(TransactionTemplate tt) {
    tt.run(manager -> {
        final Author a1 = QueryUtil.findByBusinessKey(manager,
            Author.class, "key", new AuthorKey("Wirth", "Niklaus"));
        System.out.println(a1);
        final Author a2 = QueryUtil.findByBusinessKey(manager,
            Author.class, "key", new AuthorKey("Meyer", "Bertrand"));
        System.out.println(a2);
    });
}
```


5.4 Embeddable

Sei folgendes Schema gegeben:

create.sql

```
create table AUTHOR (  
    ID integer generated by default as identity (start with 1),  
    NAME varchar (128) not null,  
    STREET varchar (64),  
    CITY varchar (64),  
    ZIP varchar (32),  
    primary key (ID),  
    unique (NAME)  
);
```

Da möglicherweise auch andere Tabellen eine `STREET`, eine `CITY`- und eine `ZIP`-Spalte haben, könnte man sich entschließen, zunächst eine Klasse `Address` zu definieren:

Address

```
package domain;  
// ...  
@Embeddable  
public class Address {  
  
    private String street;  
    private String city;  
    private String zip;  
  
    // Konstruktoren, getter, setter, toString...  
}
```

Man beachte die `@Embeddable`-Annotation!

Die Klasse `Author` kann dann unter Zuhilfenahme dieser `Address`-Klasse definiert werden:

Author

```
package domain;  
// ...  
@Entity  
public class Author {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;
```

```
@Basic
private String name;

@Embedded
private Address address;

// Konstruktoren, getter, setter, toString...
}
```

Man beachte die `@Embedded`-Annotation beim `author`-Attribut. (Der Typ eines mit `@Embedded` annotierten Attributs muss `@Embeddable` sein.)

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Author("Niklaus Wirth",
            new Address("Bernergasse 3", "Zürich", "CH-8000-8099")));
        manager.persist(new Author("Bjarne Stroustrup",
            new Address("Åboulevard", "Aarhus", "DK-8210")));
    });
}
```

```
static void demoQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select a from Author a";
        final TypedQuery<Author> query =
            manager.createQuery(jpql, Author.class);
        final List<Author> authors = query.getResultList();
        authors.forEach(author -> {
            System.out.println(author);
            System.out.println("\t" + author.getAddress());
        });
    });
}
```

5.5 Embeddable – ColumnNames

Nicht nur ein `Author`, auch ein `Publisher` kann eine Adresse haben:

create.sql

```
create table PUBLISHER (  
    ID integer generated by default as identity (start with 1),  
    NAME varchar (128) not null,  
    STREETNAME varchar (64),  
    TOWN varchar (64),  
    ZIPCODE varchar (32),  
    primary key (ID),  
    unique (NAME)  
);  
  
create table AUTHOR (  
    ID integer generated by default as identity (start with 1),  
    NAME varchar (128) not null,  
    STREET varchar (64),  
    CITY varchar (64),  
    ZIP varchar (32),  
    primary key (ID),  
    unique (NAME)  
);
```

Auch hier soll wieder die im letzten Abschnitt definierte `Address`-Klasse genutzt werden – und zwar sowohl für die Klasse `Publisher` als auch für die `Author`-Klasse.

Während die Namen `Adress`-Attribute der `AUTHOR`-Tabelle identisch sind mit den Namen den Attributen der `Address`-Klasse, ist dies bei der `PUBLISHER`-Tabelle nicht der Fall.

Innerhalb der `Publisher`-Klasse muss daher ein Spezial-Mapping definiert werden (an der im letzten Abschnitt benutzten `Author`-Klasse hat sich nichts geändert):

Publisher

```
package domain;  
// ...  
@Entity  
public class Publisher {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    @Basic  
    private String name;  
  
    @Embedded
```

```
@AttributeOverrides({
    @AttributeOverride(
        name="street", column=@Column(name="STREETNAME")),
    @AttributeOverride(
        name="city", column=@Column(name="TOWN")),
    @AttributeOverride(
        name="zip", column=@Column(name="ZIPCODE")),
})
private Address address;

// Konstruktoren, getter, setter, toString...
}
```

5.6 ElementCollection – 1

Im Kapitel "Assoziationen" werden u.a. 1:N-Verbindungen behandelt werden. Dort wird es um Beziehungen zwischen "selbständigen" Objekten gehen – um Beziehungen zwischen Objekten, von denen jedes eine eigene "Identität" hat.

Zuweilen existieren aber auch 1:N-Beziehungen, in denen nur das "Vater"-Objekt eine eigene Identität hat – und die "Kind"-Objekte bloße "Anhängsel" des Vater-Objekts sind und also eben keine eigenständige Identität hat. Sie sind nur "Teile" eines "Ganzen". Um eben solche Beziehungen geht's in diesem Abschnitt.

Zu einem Buch kann es mehrere Auflagen geben. Eine Auflage ist durch das Erscheinungsjahr und einen Was-Ist-Neu-Text beschrieben. Wir bauen eine Tabelle `BOOK` und eine Tabelle `BOOK_EDITIONS`: mit den Spalten `YEAR` und `NEWS`:

create.sql

```
create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (13) not null,
    TITLE varchar (64),
    PRICE integer,
    primary key (ID),
    unique (ISBN)
);

create table BOOK_EDITIONS (
    BOOK_ID integer not null,
    YEAR integer,
    NEWS varchar(64),
    foreign key (BOOK_ID) references BOOK
)
```

`BOOK_EDITIONS` hat zwar einen Fremdschlüssel (`BOOK_ID`), der die Tabelle `BOOK` referenziert – `BOOK_EDITIONS` hat aber keinen eigenen Primary Key.

Die Klasse `Book` ist eine `@Entity`-Klasse; die Klasse `Edition` ist KEINE(!) `@Entity`, sondern ist `@Embeddable`:

Edition

```
package domain;
// ...
import javax.persistence.Embeddable;

@Embeddable
public class Edition {
```

```
@Basic
private int year;

@Basic
private String news;

// Konstruktoren, getter, settter, toString...
}
```

Die Klasse `Book` ist eine `@Entity`-Klasse und definiert eine Liste von `Editions` (editions), die als `@ElementCollection` annotiert ist:

Book

```
package domain;
// ...
import javax.persistence.ElementCollection;

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    @ElementCollection
    // @CollectionTable(name="BOOK_EDITIONS",
    //      joinColumns = @JoinColumn(name = "BOOK_ID"))
    private List<Edition> editions;

    Book() {
    }

    public Book(String isbn, String title, double price,
                List<Edition> editions) {
        // ...
    }

    // getter, setter, toString...
}
```

Application

Die `Application`-Klasse enthält eine Helper-Methode, welche alle `Books` selektiert und diese `Books` zusammen mit ihren jeweiligen `Editions` ausgibt (die Ermittlung und Ausgabe der Bücher läuft in einer eigenen Transaktion):

```
static void showBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select b from Book b";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.getResultList().forEach(book -> {
            System.out.println(book);
            book.getEditions().forEach(
                c -> System.out.println("\t" + c));
        });
    });
}
```

Die `Persist`-Transaktion persistiert zwei `Books` – das erste `Book` hat zwei `Editions`, das zweite `Book` hat nur eine einzige `Edition`. Dabei reicht es aus, die `Editions` den `Books` zuzuweisen – sie werden automatisch zusammen mit dem jeweiligen `Book` persistiert. Nach der Persistierung werden mittels `showBooks` alle Bücher ausgegeben:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10, Arrays.asList(
            new Edition(1989, "bug free"),
            new Edition(1990, "pictures"))));
        manager.persist(new Book("2222", "Modula", 20, Arrays.asList(
            new Edition(1991, "graphics"))));
    });
    showBooks(tt);
}
```

Die Ausgaben:

```
Book [1, 1111, 10.0, Pascal]
    Edition [bug free, 1989]
    Edition [pictures, 1990]
Book [2, 2222, 20.0, Modula]
    Edition [graphics, 1991]
```

Die `demoUpdate`-Methode ermittelt das erste `Book` und fügt der `editions`-Liste dieses `Books` eine neue `Edition` hinzu. Diese wird automatisch persistiert:

```
static void demoUpdate(TransactionTemplate tt) {
    tt.run(manager -> {
        Book b = manager.find(Book.class, 1);
        b.getEditions().add(new Edition(1999, "more bugs"));
    });
}
```

```

        showBooks(tt);
    }

```

Die Ausgaben:

```

Book [1, 1111, 10.0, Pascal]
    Edition [bug free, 1989]
    Edition [pictures, 1990]
    Edition [more bugs, 1999]
Book [2, 2222, 20.0, Modula]
    Edition [graphics, 1991]

```

Die Methode `demoDelete` ermittelt das zweite `Book` und löscht die entsprechende `BOOK`-Zeile in der Datenbank:

```

static void demoDelete(TransactionTemplate tt) {
    tt.run(manager -> {
        Book b = manager.find(Book.class, 2);
        manager.remove(b);
    });
    showBooks(tt);
}

```

Die Ausgaben:

```

Book [1, 1111, 10.0, Pascal]
    Edition [bug free, 1989]
    Edition [pictures, 1990]
    Edition [more bugs, 1999]

```

Bei Entfernen des `Books` werden automatisch auch alle `BOOK_EDITION`-Zeilen gelöscht, die zu diesem `Book` gehörten. Hier der Zustand der Datenbank nach Ausführung der drei obigen Methoden:

```

BOOK
ID ISBN TITLE  PRICE
-----
1  1111 Pascal  10
-----

```

```

BOOK_EDITIONS
BOOK_ID EYEAR NEWS
-----
1       1989 bug free
1       1990 pictures
1       1999 more bugs
-----

```


5.7 ElementCollection – 2

Die `Editions` waren "komplexe" Anhängsel – eine `Edition` hat ein `year`- und ein `news`-Attribut. Zuweilen wollen wir als Anhängsel aber einfach nur eine List von Strings speichern.

Wir wollen zu einem `Book` eine Reihe von "Inhalten" speichern können. Ein Inhalt ist ein einfacher `String`. Auch dann benötigen wir natürlich eine Tabelle, in welcher die Strings gespeichert sind – für diese Tabelle ist jedoch keine eigene Java-Klasse mehr erforderlich.

create.sql

```
create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (13) not null,
    TITLE varchar (64),
    PRICE integer,
    primary key (ID),
    unique (ISBN)
);

create table BOOK_CONTENTS (
    BOOK_ID integer,
    CONTENTS varchar(64),
    foreign key (BOOK_ID) references BOOK
)
```

Auch die `BOOK_CONTENTS`-Tabelle kommt ohne Primary Key aus – auch sie besitzt natürlich einen Fremdschlüssel, der die `BOOK`-Tabelle referenziert.

Book

Die Klasse `Book` definiert ein `@ElementCollection`-Attribut namens `contents` vom Typ `List<String>` - und einen Konstruktor, dem eine Liste von Strings übergeben wird:

```
package domain;
// ...
import javax.persistence.ElementCollection;

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;
```

```

@Basic
private String title;

@Basic
private double price;

@ElementCollection
// @CollectionTable(name="BOOK_CONTENTS",
//      joinColumns = @JoinColumn(name = "BOOK_ID"))
private List<String> contents;

Book() {
}

public Book(String isbn, String title, double price,
            List<String> contents) {
    // ...
}

// getter, setter, toString...
}

```

Application

Auch hier verwendet wird eine Helper-Methode, welche alle `Books` mit ihren jeweiligen Inhalten ausgibt:

```

static void showBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select b from Book b";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.getResultList().forEach(book -> {
            System.out.println(book);
            book.getContents().forEach(
                c -> System.out.println("\t" + c));
        });
    });
}

```

Die `demoPersist`-Methode persistiert drei `Books`, von denen jedes zwei Inhalte hat – und ruft anschließend `showBooks` auf:

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10,
            Arrays.asList("Programming", "Structures")));
        manager.persist(new Book("2222", "Modula", 20,
            Arrays.asList("Programming", "Modular")));
        manager.persist(new Book("3333", "Oberon", 20,
            Arrays.asList("Programming", "Object-Orientated")));
    });
    showBooks(tt);
}

```

```
}
```

Die Ausgaben:

```
Book [1, 1111, 10.0, Pascal]
    Programming
    Structures
Book [2, 2222, 20.0, Modula]
    Programming
    Modular
Book [3, 3333, 20.0, Oberon]
    Programming
    Object-Orientated
```

`demoUpdate` ermittelt die ersten beiden Books und fügt jedem dieser beiden Books einen neuen Inhalt hinzu (die Persistierung dieser Inhalte erfolgt automatisch):

```
static void demoUpdate(TransactionTemplate tt) {
    tt.run(manager -> {
        Book b1 = manager.find(Book.class, 1);
        b1.getContents().add("Learning");
        Book b2 = manager.find(Book.class, 1);
        b2.getContents().add("Simple");
    });
    showBooks(tt);
}
```

Die Ausgaben:

```
Book [1, 1111, 10.0, Pascal]
    Programming
    Structures
    Learning
Book [2, 2222, 20.0, Modula]
    Programming
    Modular
    Simple
Book [3, 3333, 20.0, Oberon]
    Programming
    Object-Orientated
```

`demoDelete` löscht das dritte Book (natürlich werden dann automatisch auch die Inhalte dieses Books gelöscht):

```
static void demoDelete(TransactionTemplate tt) {
    tt.run(manager -> {
        Book b = manager.find(Book.class, 3);
        manager.remove(b);
    });
    showBooks(tt);
}
```

```
}
```

Die Ausgaben:

```
Book [1, 1111, 10.0, Pascal]
      Programming
      Structures
      Learning
Book [2, 2222, 20.0, Modula]
      Programming
      Modular
      Simple
```

Der Zustand der Datenbank nach Ausführung der drei Methoden:

```
BOOK
ID ISBN TITLE  PRICE
-----
1  1111 Pascal  10
2  2222 Modula  20
-----
```

```
BOOK_CONTENTS
BOOK_ID CONTENTS
-----
1      Programming
1      Structures
1      Learning
2      Programming
2      Modular
2      Simple
-----
```

5.8 Secondary Tables

JPA erlaubt die Abbildung mehrerer Tabellen auf eine einzige Klasse.

Sei etwa folgendes Schema gegeben (was in dieser einfachen Form natürlich recht "künstlich" erscheint):

create.sql

```
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    ISBN varchar (20) not null,  
    primary key (ID),  
    unique (ISBN)  
);  
  
create table BOOK_TITLE (  
    ID integer,  
    TITLE varchar (128) not null,  
    primary key (ID)  
);  
  
create table BOOK_PRICE (  
    ID integer,  
    PRICE double not null,  
    primary key (ID)  
);
```

Aus irgendwelchen Gründen sind die Daten eines Buches also auf drei Tabellen verteilt. Man möchte diese Tabellen aber abbilden auf eine einzige Klasse. Dies geschieht mit den Annotations `@SecondaryTables` und `@SecondaryTable` und mittels des `table`-Attributes der `@Column`-Annotation:

Book

```
package domain;  
// ...  
@Entity  
@Table(name = "BOOK")  
@SecondaryTables({  
    @SecondaryTable(name = "BOOK_TITLE"),  
    @SecondaryTable(name = "BOOK_PRICE")  
})  
public class Book {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    @Basic
```

```

private String isbn;

@Basic
@Column(table = "BOOK_TITLE")
private String title;

@Basic
@Column(table = "BOOK_PRICE")
private double price;

// Konstruktoren, getter, setter, toString...
}

```

Der Anwendung sieht man die Aufteilung der Daten natürlich nicht an:

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
        manager.persist(new Book("2222", "Modula", 20));
        manager.persist(new Book("3333", "Oberon", 30));
    });
}

```

Die Ausgaben:

```

Hibernate: insert into BOOK (id, isbn) values (default, ?)
Hibernate: insert into BOOK_PRICE (price, id) values (?, ?)
Hibernate: insert into BOOK_TITLE (title, id) values (?, ?)
Hibernate: insert into BOOK (id, isbn) values (default, ?)
Hibernate: insert into BOOK_PRICE (price, id) values (?, ?)
Hibernate: insert into BOOK_TITLE (title, id) values (?, ?)
Hibernate: insert into BOOK (id, isbn) values (default, ?)
Hibernate: insert into BOOK_PRICE (price, id) values (?, ?)
Hibernate: insert into BOOK_TITLE (title, id) values (?, ?)

```

```

static void demoQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query = manager.createQuery(
            "select b from Book b", Book.class);
        final List<Book> books = query.getResultList();
        books.forEach(System.out::println);
    });
}

```

Die Ausgaben:

```

Hibernate: select
    b0.id as id,
    b0.isbn as isbn,
    b1.price as price,

```

```
    b2.title as title
from BOOK b0
    left outer join BOOK_PRICE b1 on b0.id=b1_.id
    left outer join BOOK_TITLE b2 on b0.id=b2_.id
Book [1, 1111, 10.0, Pascal]
Book [2, 2222, 20.0, Modula]
Book [3, 3333, 30.0, Oberon]
```

5.9 Mapping mittels XML

Statt die erforderlichen Mapping-Informationen mittels Annotations direkt in der persistenten Klasse zu beschreiben, können diese Informationen auch in eine XML-Datei ausgelagert werden. Diese Datei muss in `META-INF` liegen und den Namen `orm.xml` tragen:

orm.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0"
>

  <package>domain</package>
  <entity class="Book" access="PROPERTY">
    <table name="T_BOOK" />
    <attributes>
      <id name="isbn">
        <column name="F_ISBN" />
      </id>
      <basic name="title">
        <column name="F_TITLE" />
      </basic>
      <basic name="price">
        <column name="F_PRICE" />
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

Die `<table>`- und die `<column>`-Elemente sind hier natürlich wieder optional. Sie sind nur dann erforderlich, wenn der Tabellename resp. die Spaltennamen von den Klassen- resp. Attribut-Namen abweichen. Dann wären auch die `<basic>`-Elemente nicht erforderlich.

Hier die im Beispiel verwendete `create.sql`-Datei:

create.sql

```
CREATE TABLE T_BOOK (
  F_ISBN VARCHAR (20),
  F_TITLE VARCHAR (128) NOT NULL,
  F_PRICE DOUBLE NOT NULL,
  PRIMARY KEY (F_ISBN)
);
```


Die `Book`-Klasse muss aber auf jeden Fall die `@Entity`- und `@Id`-Annotations besitzen:

Book

```
package domain;
// ...
@Entity
public class Book {

    @Id
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    // Konstruktoren, getter, setter, toString...
}
```

Da die meisten realen Anwendungen aber wahrscheinlich ausschließlich mit Annotations arbeiten, wird im folgenden auf die Verwendung der möglichen XML-Alternative des Mappings verzichtet.

6 Queries

JPA verwendet JPQL als Abfragesprache.

JPQL steht für "Java Persistence Query Language". Es handelt sich dabei um eine "objektorientierte" Abfragesprache, die von ihrer syntaktischen Struktur aber große Ähnlichkeiten mit SQL hat.

Um eine Abfrage abzusetzen, wird ein `Query`-Objekt (oder ein `TypedQuery`-Objekt) erzeugt. Auf dieses Objekt kann dann entweder `getSingleResult` oder `getResultList` aufgerufen werden.

Die Objekte, die von `getSingleResult` geliefert werden bzw. in einer `List` von `getResultList` geliefert werden, können unterschiedlicher Natur sein:

- Objekte der persistenten Klassen (der `@Entity`-Klassen). Alle obigen Anfragen haben `Book`-Objekte oder `List<Book>`-Objekte geliefert.
- einfache POJOs (deren Klassen nicht als `@Entity` definiert sind - im Falle der sog. "Constructor-Queries")
- Arrays einfacher Werte (wenn im JPQL-String eine Projektion definiert ist)
- skalare Werte (wenn das Resultat der Abfrage aus einer Zeile und einer Spalte besteht).

Nur solche Objekte, deren Klassen als `@Entity` definiert sind, werden im Cache hinterlegt. Die Ergebnisse einer Constructor-Query allerdings werden auch dann nicht gecached, wenn eine solche Anfrage `@Entity`-Objekte erzeugt.

Den Anwendungsbeispielen liegt die altbekannte `Book`-Klasse zugrunde (wobei als Primary Key eine generierte ID verwendet wird).

6.1 Einfache Queries

Im folgenden wird gezeigt, wie mittels `createQuery` typsichere `TypedQuery`-Objekte erzeugt werden können. Auf solche Objekte können dann u.a. die Methoden `getResultList` und `getSingleResult` aufgerufen werden.

create.sql

```
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    ISBN varchar (20) not null,  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    primary key (ID),  
    unique (ISBN)  
);
```

Application

Die `demo`-Methoden benutzen zwei Helper-Methoden. Beide Methoden sind mit einem JPQL-String parametrisiert. Die Methode `showBooks` führt einen Mengen-Select aus (mittels `TypedQuery.getResultList`), die Methode `showBook` führt einen Einzelsatz-Select aus (mittels `TypedQuery.getSingleResult`):

```
private static void showBooks(EntityManager manager, String jpql) {  
    System.out.println("==> " + jpql);  
    final TypedQuery<Book> query =  
        manager.createQuery(jpql, Book.class);  
    final List<Book> books = query.getResultList();  
    books.forEach(System.out::println);  
    System.out.println();  
}
```

```
private static void showBook(EntityManager manager, String jpql) {  
    System.out.println("==> " + jpql);  
    final TypedQuery<Book> query =  
        manager.createQuery(jpql, Book.class);  
    final Book book = query.getSingleResult();  
    System.out.println(book);  
    System.out.println();  
}
```

Wie persistieren fünf Bücher:

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        manager.persist(new Book("1111", "Pascal", 10));  
        manager.persist(new Book("2222", "Modula", 20));  
        manager.persist(new Book("3333", "Oberon", 30));  
    });  
}
```

```
        manager.persist(new Book("4444", "Eiffel", 40));
        manager.persist(new Book("5555", "Simula", 40));
    });
}
```

demoQuery1 führt Mengen-Selects aus:

```
static void demoQueryResultList(TransactionTemplate tt) {
    tt.run(manager -> {
        showBooks(manager,
            "select b from Book b");
        showBooks(manager,
            "select b from Book b where b.price < 30");
        showBooks(manager,
            "select b from Book b where b.title like '%a%'");
    });
}
```

Die Ausgaben:

```
==> select b from Book b
Book [1, 1111, 10.0, Pascal]
Book [2, 2222, 20.0, Modula]
Book [3, 3333, 30.0, Oberon]
Book [4, 4444, 40.0, Eiffel]
Book [5, 5555, 40.0, Simula]
```

```
==> select b from Book b where b.price < 30
Book [1, 1111, 10.0, Pascal]
Book [2, 2222, 20.0, Modula]
```

```
==> select b from Book b where b.title like '%a%'
Book [1, 1111, 10.0, Pascal]
Book [2, 2222, 20.0, Modula]
Book [5, 5555, 40.0, Simula]
```

Die folgende Methode führt Einzelsatz-Zugriffe aus:

```
static void demoQuerySingleResult(TransactionTemplate tt) {
    tt.run(manager -> {
        showBook(manager,
            "select b from Book b where b.isbn = '1111'");
        showBook(manager,
            "select b from Book b where b.isbn = '4444'");
    });
}
```

Die Ausgaben:

```
==> select b from Book b where b.isbn = '1111'
Book [1, 1111, 10.0, Pascal]
```

```
==> select b from Book b where b.isbn = '4444'  
Book [4, 4444, 40.0, Eiffel]
```

Die folgende zeigt das Verhalten von `getSingleResult` bei einer leeren Ergebnismenge:

```
static void demoQueryWithNoResultException(  
    TransactionTemplate tt) {  
    tt.run(manager -> {  
        final TypedQuery<Book> query = manager.createQuery(  
            "select b from Book b where b.isbn = '1234',  
            Book.class);  
        try {  
            final Book book = query.getSingleResult();  
            System.out.println(book);  
        }  
        catch (NoResultException e) {  
            System.out.println("This exception is expected:");  
            System.out.println(e);  
        }  
    });  
}
```

Die Ausgaben:

```
This exception is expected:  
javax.persistence.NoResultException:  
    No entity found for query
```

Und die folgende Methode zeigt das Verhalten von `getSingleResult` bei einer mehrzeiligen Ergebnismenge:

```
static void demoQueryWithNonUniqueResultException(  
    TransactionTemplate tt) {  
    tt.run(manager -> {  
        final TypedQuery<Book> query = manager.createQuery(  
            "select b from Book b where b.title like '%a%'",  
            Book.class);  
        try {  
            final Book book = query.getSingleResult();  
            System.out.println(book);  
        }  
        catch (NonUniqueResultException e) {  
            System.out.println("This exception is expected:");  
            System.out.println(e);  
        }  
    });  
}
```

Die Ausgaben:

```
This exception is expected:  
javax.persistence.NonUniqueResultException:
```

result returns more than one elements

`EntityManager.getResultList` liefert ein `List`-kompatibles Objekt zurück (die `List` ist dabei natürlich möglicherweise leer). `getSingleResult` liefert bei genau einem Treffer das entsprechende Objekt zurück. Bei keinem Treffer wird eine `NoResultException` geworfen. Bei mehr als einem Treffer wird eine `NonUniqueResultException` geworfen. `getSingleResult` liefert also niemals `null`.

Das Interface `EntityManager` enthält u.a. folgende zwei `createQuery`-Methoden:

```
public interface EntityManager {  
    public Query createQuery(String jpql);  
    public <T> TypedQuery<T> createQuery(String jpql, Class<T> cls);  
    //...  
}
```

Im obigen Beispiel wurde die zweite Methode verwendet – die typsichere Variante.

Hier ein Ausschnitt aus dem Interface `Query`:

```
public interface Query<T> {  
    public List getResultList();  
    public Object getSingleResult();  
    // ...  
}
```

Das Resultat von `getResultList` muss dann z.B. auf `List<Book>` gecastet werden; das Resultat von `getSingleResult` auf `Book`. Diese Downcasts sind bei der Verwendung der typsicheren Variante nicht mehr erforderlich.

Hier ein Ausschnitt aus dem Interface `TypedQuery`:

```
public interface TypedQuery<T> {  
    public List<T> getResultList();  
    public T getSingleResult();  
    // ...  
}
```

Wird `getResultList` z.B. auf eine `TypedQuery<Book>`-Referenz aufgerufen, liefert diese Methode eine `List<Book>` (und nicht nur `List`). Wird `getSingleResult` auf eine solche Referenz aufgerufen, wird `Book` geliefert (und nicht nur `Object`).

6.2 Parametrisierte Queries

Anstatt z.B. in einer `where`-Klausel von JPQL Werte fest einzubinden (per Konkatenation), sollte es möglich sein, QL-Statements mit Platzhaltern zu versehen.

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
        manager.persist(new Book("2222", "Modula", 20));
        manager.persist(new Book("3333", "Oberon", 30));
        manager.persist(new Book("4444", "Eiffel", 40));
    });
}
```

```
static void demoQueryNamedParameter(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query1 = manager.createQuery(
            "select b from Book b where b.price < :price",
            Book.class);
        query1.setParameter("price", 30.0);
        query1.getResultList().forEach(System.out::println);
    });
}
```

Benamste Platzhalter werden in einem JPQL-String mit einem ":" eingeleitet. Die Platzhalter werden dann über die `TypedQuery`-Methode `setParameter(String name)` durch konkrete Werte ersetzt.

Die Ausgaben:

```
Book [1, 1111, 10.0, Pascal]
Book [2, 2222, 20.0, Modula]
```

```
static void demoQueryPositionParameter(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query2 = manager.createQuery(
            "select b from Book b where b.title like ?",
            Book.class);
        query2.setParameter(0, "%a%");
        query2.getResultList().forEach(System.out::println);
    });
}
```

Positions-Parameter werden mit einem "?" gekennzeichnet. Die Platzhalter werden dann über die `TypedQuery`-Methode `setParameter(int index)` durch konkrete Werte ersetzt (wobei die Indizes mit 0 beginnen (also anders als bei JDBC!)).

Die Ausgaben:

```
Book [1, 1111, 10.0, Pascal]  
Book [2, 2222, 20.0, Modula]
```

Benannte Platzhalter haben u.a. folgenden Vorteil gegenüber den Positionsparametern: ein und derselbe JPQL-String kann an mehreren Stellen Platzhalter gleichen Namens haben. Und Namen sind besser lesbar...

6.3 Projection

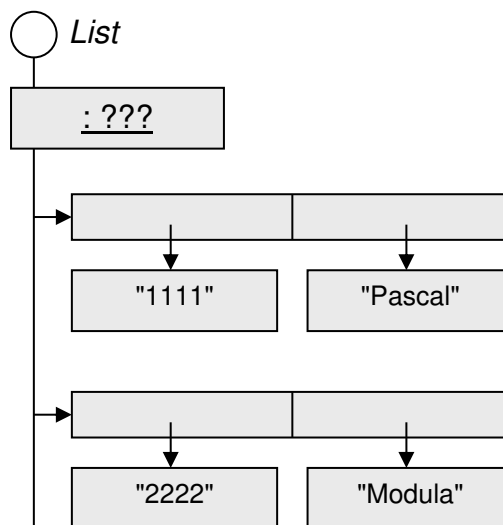
Bislang lieferte die `Query`-Methode (bzw. mit der `TypedQuery`-Methode) `getResultList` stets eine Liste von Objekten persistenter Klassen (`Book`, `Author`, ...). In JPQL wurde eben deshalb auch keine Spaltenliste ("Projektion") verwendet - denn zur Produktion eines Objekts einer persistenten Klasse muss ohnehin für jedes Attribut dieser Klasse die entsprechende Tabellenspalte gelesen werden.

Aber auch bei JPQL kann eine Projektion verwendet werden. Dann liefert JPA aber keine Liste von Objekten persistenter Klassen, sondern einfach nur eine Liste von `Object`-Arrays. Hat man per Projektion z.B. zwei Spalten ausgewählt, erzeugt JPA eine Liste von Arrays, die jeweils zwei Werte beinhalten: die gelesenen Spaltenwerte in `Object`-kompatibler Form (als `String`-, `Integer`-, `Double`- Objekte etc.)

Wird z.B. folgender `select` ausgeführt:

```
select b.isbn, b.title from Book b order by b.isbn
```

Dann wird `getResultList` z.B. folgende Liste liefern:



Die Namen, die in der Projektion verwendet werden, sind natürlich nicht die Namen der Tabellenspalten, sondern die Namen der entsprechenden Attribute der Java-Klasse.

Man beachte, dass ein solches Resultat nicht in den Cache eingefügt wird (dort existieren nur Objekte persistenter Klassen!).

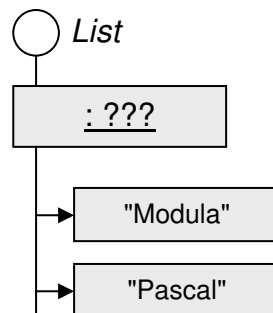
Wenn statt `getResultList` Methode `getSingleResult` aufgerufen würde, dann würde diese - im Erfolgsfall - die Referenz auf genau ein einziges Array-Objekt zurückliefern.

Hinweis: Mit dem Interface `TypedQuery` kann im Falle, dass eine `List` von Arrays geliefert wird, nicht(!) gearbeitet werden. Man muss sich hier mit der "unsicheren" `Query`-Variante begnügen.

Was passiert bei folgendem `select`:

```
select b.title from Book b order by b.title
```

Hier das Resultat:



Wenn die Projektionsliste nur aus einem Element besteht, wird also keine Liste von `Object`-Arrays geliefert, sondern eine Liste von `Objects`, welche direkt die Spaltenwerte repräsentieren (hier z.B.: eine Liste von `Strings`). Hier kann wieder `TypedQuery` genutzt werden.

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
        manager.persist(new Book("2222", "Modula", 20));
    });
}
```

```
static void demoQuery1(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Object[]> query = manager.createQuery(
            "select b.isbn, b.title from Book b order by b.isbn",
            Object[].class);
        final List<Object[]> rows = query.getResultList();
        for (final Object[] row : rows) {
            for (final Object value : row)
                System.out.println(value);
            System.out.println();
        }
    });
}
```

Die Ausgaben:

Hibernate: select isbn, title Book isbn

1111
Pascal

2222
Modula

```
static void demoQuery2(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final TypedQuery<String> query = manager.createQuery(  
            "select b.title from Book b order by b.title",  
            String.class);  
        final List<String> titles = query.getResultList();  
        for (final String title : titles) {  
            System.out.println(title);  
        }  
    });  
}
```

Die Ausgaben:

Hibernate: select title Book order by title

Modula
Pascal

6.4 Projection – Performance

Das Erstellen von Listen von Arrays ist wesentlich performanter als die Erzeugung "richtiger" Entity-Objekte. Hierzu folgender Test:

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        for (int i = 0; i < 500; i++)  
            manager.persist(new Book("1111" + i, "Pascal", 10));  
    });  
}
```

```
static void demoPerformance(TransactionTemplate tt) {  
    tt.runPerformanceTest(500,  
        manager -> {  
            final String jpql =  
                "select b from Book b";  
            final TypedQuery<Book> query =  
                manager.createQuery(jpql, Book.class);  
            query.getResultList();  
        },  
        manager -> {  
            final String jpql =  
                "select b.isbn, b.title, b.price from Book b";  
            final TypedQuery<Object[]> query =  
                manager.createQuery(jpql, Object[].class);  
            query.getResultList();  
        }  
    );  
}
```

Die Ausgaben:

```
duration[0] = 2233 milliseconds  
duration[1] = 951 milliseconds
```

Arrays von Objects zu erzeugen ist also mehr als doppelt so performant wie die Erzeugung von Entities.

6.5 Utility-Klassen: Row und RowList

Im letzten Abschnitt wurde die Möglichkeit dargestellt, Tabellenzeilen zu einfachen Arrays von `Objects` transformieren zu lassen. Das spart Kosten: weder muss Reflection bemüht werden noch wird der Cache belastet. Und schließlich kann man über die zurückgelieferten Arrays auf einfache Weise iterieren (über "richtige" Objekte kann man bekanntlich nur per Reflection "iterieren").

Dieses Herangehen kann offenbar immer dann sinnvoll sein, wenn es ausschließlich um die Präsentation von Daten geht – z.B. mittels einer `Swing-JTable` oder einer HTML-Tabelle. Wenn dagegen die Daten individuell in jeweils speziellen Fachlogiken bearbeitet werden sollen (was neben dem Lesen natürlich auch Schreiben einschließt), ist die Produktion "richtiger" Objekte der bessere Weg.

Im folgenden werden einige kleine Hilfsklassen entwickelt, welche den Umgang mit den Ergebnissen solchen Projektions-Queries vereinfachen. Insbesondere abstrahieren diese Klassen das "Problem", dass als Ergebnisse solcher Queries i.d.R. Listen von Arrays geliefert werden, manchmal (wenn die Projektion nur eine Spalte umfasst) aber Listen einfacher `Objects` (siehe die Ergebnisse der Queries des letzten Abschnitts).

Eine `Row` repräsentiert die Werte einer Tabellenzeile. Dabei ist es egal, ob es sich um einen einzigen Wert oder um viele Werte handelt - auch ein einziger Wert wird behandelt als Array mit einem Element:

Row

```
package util;

public class Row {

    private final String[] names;
    private final Object[] values;

    public Row(String[] names, Object obj) {
        this.names = names;
        if (obj.getClass().isArray())
            this.values = (Object[]) obj;
        else
            this.values = new Object[] { obj };
        if (names.length != this.values.length)
            throw new IllegalArgumentException();
    }

    public int size() {
        return this.values.length;
    }

    public String getName(int index) {
```

```

        return this.names[index];
    }

    public Object getValue(int index) {
        return this.values[index];
    }
}

```

Dem Konstruktor von `Row` muss entweder ein `Object[]` oder ein einfaches `Object` übergeben werden. Weiterhin muss ein Array von Strings übergeben werden, welcher die zu den gelesenen Spalten gehörigen Attribut-Namen enthält.

Eine `RowList` enthält eine `List<Row>`:

RowList

```

package util;
// ...
public class RowList {

    private final String[] names;
    private final List<Row> rows = new ArrayList<Row>();

    public RowList(List<?> list, String... names) {
        this.names = names;
        list.forEach(obj -> this.rows.add(new Row(names, obj)));
    }

    public int getColumnCount() {
        return this.names.length;
    }

    public int size() {
        return this.rows.size();
    }

    public String getName(int index) {
        return this.names[index];
    }

    public Row getRow(int index) {
        return this.rows.get(index);
    }

    public Object getValue(int rowIndex, int columnIndex) {
        return this.getRow(rowIndex).getValue(columnIndex);
    }
}

```

Dem Konstruktor muss ein Array von Attribut-Namen und eine `List` übergeben werden. Die Elemente der `List` können entweder Arrays von Objects oder einfache Objects sein. Als zweites Argument kann also stets das Resultat von `getResultList` übergeben werden.

Um eine `RowList` in einer `JTable` anzuzeigen, kann eine kleine Adapter-Klasse benutzt werden:

RowListAdapterForTableModel

```
package util;

import javax.swing.table.AbstractTableModel;

public class RowListAdapterForJTable extends AbstractTableModel {
    private static final long serialVersionUID = 1L;

    private final RowList rowList;

    public RowListAdapterForJTable(RowList rowList) {
        this.rowList = rowList;
    }

    @Override
    public String getColumnName(int columnIndex) {
        return this.rowList.getName(columnIndex);
    }

    @Override
    public int getColumnCount() {
        return this.rowList.getColumnCount();
    }

    @Override
    public int getRowCount() {
        return this.rowList.size();
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        return this.rowList.getValue(rowIndex, columnIndex);
    }
}
```

Dem Konstruktor von `RowListAdapterForJTable` wird eine `RowList` übergeben. Ein solcher Adapter kann als "Model" für eine `JTable` verwendet werden.

Die folgende Anwendung demonstriert sowohl die Verwendung von `RowLists` als auch die Verwendung dieser Adapter-Klasse:

Application

`showRowList` zeigt eine `RowList` mittels eine `JTables` an:

```
private static void showRowList(RowList list, int x, int y) {
    final JFrame frame = new JFrame();
```

```

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        final JTable table = new JTable();
        final JScrollPane scrollPane = new JScrollPane(table);
        table.setModel(new RowListAdapterForJTable(list));
        frame.add(scrollPane);
        scrollPane.setPreferredSize(new Dimension(300, 100));
        frame.pack();
        frame.setLocation(x, y);
        frame.setVisible(true);
    }

```

Wir persistieren zwei Bücher:

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
        manager.persist(new Book("2222", "Modula", 20));
    });
}

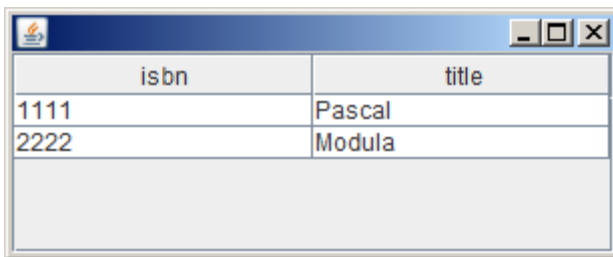
```

```

static void demoQuery1(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Object[]> query = manager.createQuery(
            "select b.isbn, b.title from Book b order by b.isbn",
            Object[].class);
        final List<Object[]> rows = query.getResultList();
        final RowList list = new RowList(rows, "isbn", "title");
        showRowList(list, 100, 100);
    });
}

```

Die obige Methode liefert folgende GUI:



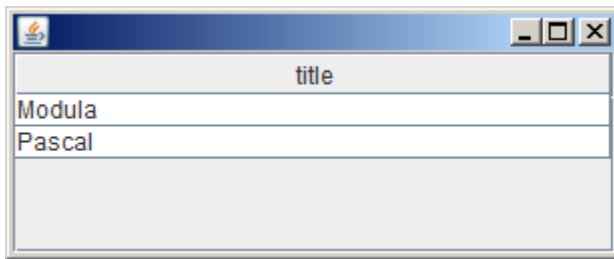
isbn	title
1111	Pascal
2222	Modula

```

static void demoQuery2(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<String> query = manager.createQuery(
            "select b.title from Book b order by b.title",
            String.class);
        final List<String> titles = query.getResultList();
        final RowList list = new RowList(titles, "title");
        showRowList(list, 300, 300);
    });
}

```

Die obige Methode liefert folgende GUI:



6.6 Constructor Expressions

Neben der `@Entity`-Klasse `Book` wird im folgenden eine ganz "dumme" Klasse benutzt, eine Klasse, die nicht(!) als `@Entity` gekennzeichnet ist - und somit keine(!) persistente Klasse ist. Objekte dieser Klasse sind einfach nur "komplexe Werte". Man beachte, dass sie keinen parameterlosen Konstruktor besitzt (und auch keine getter / setter benötigt).

Sie besitzt stattdessen drei unterschiedlich parametrisierte Konstruktoren:

BookData

```
package appl;

import common.util.Util;

public class BookData {

    private String isbn;
    private String title;
    private Double price;

    public BookData(String isbn, String title, Double price) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
    }

    public BookData(String isbn, String title) {
        this(isbn, title, null);
    }

    public BookData(String title) {
        this(null, title, null);
    }

    // getter, setter, toString...
}
```

Dann kann ein JPQL-String z.B. wie folgt aufgebaut sein:

```
select new appl.BookData(b.isbn, b.title, b.price) from Book b
```

Für jede gelesene Tabellenzeile wird nun ein `BookData`-Objekt erzeugt und mittels eines in der `BookData`-Klasse definierten Konstruktors(!) initialisiert (im obigen Beispiel mit dem ersten Konstruktor dieser Klasse). Im Gegensatz zum "normalen" Verfahren findet die Initialisierung hier also nicht über setter-Methoden statt.

Objekte, die auf solche Weise erzeugt werden, werden nicht(!) im Cache abgestellt.

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        manager.persist(new Book("1111", "Pascal", 10));  
        manager.persist(new Book("2222", "Modula", 20));  
        manager.persist(new Book("3333", "Oberon", 30));  
    });  
}
```

Die ersten drei der folgenden Methoden nutzen die Konstruktoren der `BookData`-Klasse.

```
static void demoQueryThree(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql =  
            "select new appl.BookData(b.isbn, b.title, b.price) " +  
            "from Book b";  
        final TypedQuery<BookData> query =  
            manager.createQuery(jpql, BookData.class);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

Die Ausgaben:

```
Hibernate: select isbn, title, price from Book  
BookData [1111, 10.0, Pascal]  
BookData [2222, 20.0, Modula]  
BookData [3333, 30.0, Oberon]
```

```
static void demoQueryTwo(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql =  
            "select new appl.BookData(b.isbn, b.title) from Book b";  
        final TypedQuery<BookData> query =  
            manager.createQuery(jpql, BookData.class);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

Die Ausgaben:

```
Hibernate: select isbn, title, from Book  
BookData [1111, null, Pascal]  
BookData [2222, null, Modula]  
BookData [3333, null, Oberon]
```

```
static void demoQueryOne(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql =  
            "select new appl.BookData(b.title) from Book b";  
        final TypedQuery<BookData> query =
```

```

        manager.createQuery(jpql, BookData.class);
        query.getResultList().forEach(System.out::println);
    });
}

```

Die Ausgaben:

```

Hibernate: select title, from Book
BookData [null, null, Pascal]
BookData [null, null, Modula]
BookData [null, null, Oberon]

```

Statt der `BookData`-Klasse kann auch die Klasse `Book` in einer Constructor-Expression verwendet werden. Die aufgrund einer solchen Anfrage erzeugten `Book`-Objekte sind dann aber ebenfalls nicht(!) im Cache enthalten. Die Klasse `Book` wird dann ebenfalls einfach als "dumme Klasse" behandelt – von ihrer `@Entity`-Qualität wird abgesehen:

```

static void demoQueryEntityAsNonEntity(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select new domain.Book(b.isbn, b.title, b.price) " +
            "from Book b";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.getResultList().forEach(book -> {
            System.out.println(book);
            System.out.println(manager.contains(book));
        });
    });
}

```

Die Ausgaben:

```

Hibernate: select isbn, title, price from Book
Book [null, 1111, 10.0, Pascal]
false
Book [null, 2222, 20.0, Modula]
false
Book [null, 3333, 30.0, Oberon]
false

```

6.7 Constructor-Expressions – Performance

Auch hierzu ein Performance-Test (im "Performance"-Projekt):

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        for (int i = 0; i < 500; i++)  
            manager.persist(new Book("1111" + i, "Pascal", 10));  
    });  
}
```

```
static void demoPerformance(TransactionTemplate tt) {  
    tt.runPerformanceTest(500,  
        manager -> {  
            final String jpql =  
                "select b from Book b";  
            final TypedQuery<Book> query =  
                manager.createQuery(jpql, Book.class);  
            query.getResultList();  
        },  
        manager -> {  
            final String jpql =  
                "select new domain.Book(" +  
                "b.isbn, b.title, b.price) from Book b";  
            final TypedQuery<Book> query =  
                manager.createQuery(jpql, Book.class);  
            query.getResultList();  
        }  
    );  
}
```

Die Ausgaben:

```
duration[0] = 2075 milliseconds  
duration[1] = 966 milliseconds
```

Man sieht: auch bei der Benutzung von Constructor-Expressions ist die Performance wesentlich besser als bei der Erzeugung "richtiger" Entities.

6.8 Aggregat-Funktionen

JPQL ermöglicht natürlich - ebenso wie auch SQL - die Benutzung von Aggregat-Funktionen: `count`, `min`, `max` etc.

Auf das `Query`-(resp. `TypedQuery`-) Objekt, welches einen entsprechenden `select` ausführen soll, wird dann natürlich die Methode `getSingleResult` aufgerufen. Diese liefert einen Array von `Objects` zurück - oder aber nur ein einziges `Object` (falls die Projektionsliste nur aus einem einzigen Element besteht).

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
        manager.persist(new Book("2222", "Modula", 30));
        manager.persist(new Book("3333", "Oberon", 20));
    });
}
```

```
static void demoCount(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select count(b) from Book b";
        final TypedQuery<Long> query =
            manager.createQuery(jpql, Long.class);
        final long count = query.getSingleResult();
        System.out.println(count);
    });
}
```

Die Ausgabe: 3

```
static void demoMinMax(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select min(b.price), max(b.price) from Book b";
        final TypedQuery<Object[]> query =
            manager.createQuery(jpql, Object[].class);
        final Object[] values = query.getSingleResult();
        System.out.println(values[0]);
        System.out.println(values[1]);
    });
}
```

Die Ausgaben: 10.0 30.0

6.9 Bulk Update / Delete

Angenommen, die Preise aller Bücher sollen um 10% erhöht werden. Man könnte folgenden Code schreiben:

```
TypedQuery<Book> query = manager.createQuery(
    "select from Book b", Book.class);
List<Book> books = query.getResultList();
for (Book b : books)
    b.setPrice(b.getPrice() * 1.1);
```

Angenommen weiterhin, die `BOOK`-Tabelle enthält 10000 Zeilen. Dann würden beim Aufruf von `getResultList` 10000 `Book`-Objekte erzeugt und in den Cache eingetragen; und beim Commit würden 10000 `UPDATE`-Befehle zur Datenbank geschickt. Nicht eben performant...

Bei Mengen-Updates (oder Mengen-Deletes) kann an `createQuery` auch eine JPQL-`update`-Anweisung (oder `delete`-Anweisung) übergeben werden, z.B.:

```
update Book b set b.price = b.price * 1.1
```

Oder:

```
delete from Book b where b.price < 40
```

(Man beachte, dass auch hier die verwendeten Namen Java-Namen sind (Klassen, Properties resp. Attribute) - und keine Datenbanknamen (Tabellen, Spalten).

Um das `Query`-Objekt dann zu veranlassen, eine entsprechende SQL-Anweisung zur Datenbank zu schicken, muss die Methode `executeUpdate` aufgerufen werden.

Application

Wir persistieren zwei Bücher:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 30));
        manager.persist(new Book("2222", "Modula", 40));
    });
}
```

Wir benutzen die `Query`-Methode `executeUpdate`, um einen Mengen-`UPDATE` zur Datenbank zu schicken:

```
static void demoUpdate(TransactionTemplate tt) {
    tt.run(manager -> {
```

```
        final String jpql =
            "update Book b set b.price = b.price * 1.1";
        final Query query = manager.createQuery(jpql);
        final int result = query.executeUpdate();
        System.out.println("result of update = " + result);
    });
}
```

Die Ausgaben:

```
Hibernate: update Book set price=price*1.1
result of update = 2
```

Wir benutzen wiederum `executeUpdate`, um einen Mengen-DELETE zur Datenbank zu schicken:

```
static void demoDelete(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "delete from Book b where b.price < 40";
        final Query query = manager.createQuery(jpql);
        final int result = query.executeUpdate();
        System.out.println("result of delete = " + result);
    });
}
```

Die Ausgaben:

```
Hibernate: delete from Book where price < 40
result of delete = 1
```

Das Resultat von `executeUpdate` ist in beiden die Anzahl der von dem ausgeführten SQL-Befehl betroffenen Datenbank-Zeilen.

6.10 Bulk Update / Delete - Performance

Ein Performance-Test:

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        for(int i = 0; i < 500; i++)
            manager.persist(new Book("1111" + i, "Pascal", 10));
    });
}
```

```
static void demoPerformance(TransactionTemplate tt) {
    tt.runPerformanceTest(100,
        manager -> {
            final String jpql =
                "select b from Book b";
            final TypedQuery<Book> query =
                manager.createQuery(jpql, Book.class);
            final List<Book> bookList = query.getResultList();
            bookList.forEach(
                book -> book.setPrice(book.getPrice() * 1.1));
        },
        manager -> {
            final String jpql =
                "update Book b set b.price = b.price * 1.1";
            final Query query = manager.createQuery(jpql);
            query.executeUpdate();
        }
    );
}
```

Die Ausgaben:

```
duration[0] = 636 milliseconds
duration[1] = 186 milliseconds
```

Der Bulk-Update ist etwa 4 mal schneller als der Single-Object-Update.

6.11 Named Queries

Die JPQL-Strings können aus der Applikation in die persistenten Klassen verlagert werden. Dort werden sie mittels Annotations definiert. Bei dieser Definition erhalten sie einen Namen, über welchen sie dann in der Applikation ansprechbar sind.

Book

Die `Book`-Klasse wird erweitert um eine `@NamedQueries`-Annotation (Plural), die beliebig viele `@NamedQuery`-Annotationen enthalten kann (Singular). Jede `@NamedQuery`-Annotation hat einen Namen und einen JPQL-String. Als Namen werden im folgenden Konstanten verwendet, die in der Klasse `Book` definiert sind – unter diesen Namen sind die Queries dann auch in der eigentlichen Applikation ansprechbar.

Die erste `@NamedQuery`-Annotation definiert einen JPQL-String, der benamste Parameter enthält; im JPQL-String der zweiten `@NamedQuery`-Annotation werden Positions-Parameter verwendet:

```
package domain;
// ...
@Entity
@NamedQueries({
    @NamedQuery(
        name = Book.PRICE_BETWEEN_1,
        query = "select b from Book b where b.price between :min and :max"
    ),
    @NamedQuery(
        name = Book.PRICE_BETWEEN_2,
        query = "select b from Book b where b.price between ? and ?"
    )
})

public class Book {

    public static final String PRICE_BETWEEN_1 = "Book.PriceBetween1";
    public static final String PRICE_BETWEEN_2 = "Book.PriceBetween2";

    // wie gehabt ...
}
```

Application

`demoPersist` persistiert vier Bücher:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10.0));
        manager.persist(new Book("2222", "Modula", 20.0));
        manager.persist(new Book("3333", "Oberon", 30.0));
    });
}
```

```

        manager.persist(new Book("4444", "Eiffel", 40.0));
    });
}

```

demoBetween1 zeigt die Verwendung der ersten @NamedQuery (benamste Parameter):

```

static void demoBetween1(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query =
            manager.createNamedQuery(
                Book.PRICE_BETWEEN_1, Book.class);
        query.setParameter("min", 20.0);
        query.setParameter("max", 30.0);
        query.getResultList().forEach(System.out::println);
    });
}

```

Die Ausgaben:

```

Book [2, 2222, 20.0, Modula]
Book [3, 3333, 30.0, Oberon]

```

demoBetween2 zeigt die Verwendung der zweiten @NamedQuery (Positions-Parameter):

```

static void demoBetween2(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query =
            manager.createNamedQuery(
                Book.PRICE_BETWEEN_2, Book.class);
        query.setParameter(0, 20.0);
        query.setParameter(1, 30.0);
        query.getResultList().forEach(System.out::println);
    });
}

```

Die Ausgaben sind dieselben wie diejenigen von demoBetween1.

Named Queries können sofort bei der Erstellung der `PersistenceManagerFactory` auf syntaktische Korrektheit geprüft werden (Fehler fallen also sofort beim Start der Anwendung auf – und nicht erst dann, wenn der Query ausgeführt wird).

Und ein Service benötigt keine Inflation von `find`-Methoden, sondern möglicherweise nur noch eine einzige:

```

public class LibraryService {
    // ...
    public List<Book> find(String queryId, Object... params) { ... }
}

```

Der `find`-Methode wird einfach eine Query-ID übergeben ein Varargs-Array mit beliebig vielen Positions-Parametern.

Hier ein möglicher Aufruf:

```
LibraryService libraryService = ...  
List<Book> libraryService.find(Book.PRICE_BETWEEN_2, 20.0, 30.0);
```

Die Bedeutung der hier übergebenen Parameter wird natürlich nur auf Grundlage des Query-Strings deutlich, der in der entsprechenden `@NamedQuery`-Annotation definiert ist (hier: `Book.PRICE_BETWEEN_2`).

6.12 Adding Named Queries

Statt Named Queries mittels Annotationen zu definieren, können solche Queries auch programmatisch erzeugt und in einer Query-Registratur der `EntityManagerFactory` eingetragen werden.

Aus der Klasse `Book` ist die `@NamedQueries`-Annotation verschwunden:

Book

```
package domain;
// ...
@Entity
public class Book { ... }
```

Application

Mittels des `EntityManagers` ermitteln wir die `EntityManagerFactory` und benutzen dessen `addNamedQuery`-Methode. Wir registrieren zwei Queries mit den Namen `"Book.PriceBetween"` und `"Book.PriceMin"`:

```
static void addNamedQueries(TransactionTemplate tt) {
    tt.run(manager -> {
        final EntityManagerFactory factory =
            manager.getEntityManagerFactory();

        final Query query1 = manager.createQuery(
            "select b from Book b " +
            "where b.price between :min and :max");
        factory.addNamedQuery("Book.PriceBetween", query1);

        final Query query2 = manager.createQuery(
            "select b from Book b where b.price >= :min");
        query2.setMaxResults(2);
        factory.addNamedQuery("Book.PriceMin", query2);
    });
}
```

Die `query2` wird zusätzlich noch ein wenig konfiguriert: mittels der Methode `setMaxResults` wird die maximale Anzahl der Treffer festgelegt.

Die `demoPersist`-Methode persistiert vier Bücher:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10.0));
        manager.persist(new Book("2222", "Modula", 20.0));
        manager.persist(new Book("3333", "Oberon", 30.0));
        manager.persist(new Book("4444", "Eiffel", 40.0));
    });
}
```

```
    });  
}
```

Die demo-Methoden benutzen auch hier die EntityManager-Methode `createNamedQuery`:

```
static void demoQueryPriceBetween(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final TypedQuery<Book> query =  
            manager.createNamedQuery("Book.PriceBetween", Book.class);  
        query.setParameter("min", 20.0);  
        query.setParameter("max", 30.0);  
        final List<Book> books = query.getResultList();  
        books.forEach(System.out::println);  
    });  
}
```

Die Ausgaben:

```
Book [2, 2222, 20.0, Modula]  
Book [3, 3333, 30.0, Oberon]
```

```
static void demoQueryPriceMin(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final TypedQuery<Book> query =  
            manager.createNamedQuery("Book.PriceMin", Book.class);  
        query.setParameter("min", 20.0);  
        final List<Book> books = query.getResultList();  
        books.forEach(System.out::println);  
    });  
}
```

Die Ausgaben (man beachte, dass die Anzahl der Treffer auf 2 beschränkt wurde...):

```
Book [2, 2222, 20.0, Modula]  
Book [3, 3333, 30.0, Oberon]
```

6.13 Native Queries

JPA erlaubt die Formulierung nativer Queries – sowohl direkt in der Applikation (`manager.createNativeQuery`) als auch als `NamedNativeQueries` (per Annotation in der persistenten Klasse).

Book

Wir erweitern die Klasse `Book` um eine `@NamedNativeQueries`-Annotation erweitert, welche ihrerseits eine `@NamedNativeQuery` enthält. Letztere hat einen Namen, einen SQL-Select-String und einen Verweis auf das `Class`-Objekt der Klasse `Book`. Als Name wird wiederum eine Konstante verwendet, die in der `Book`-Klasse definiert ist (`PRICE_BETWEEN`)

```
package domain;
// ...
@Entity
@NamedNativeQueries({
    @NamedNativeQuery(
        name = Book.PRICE_BETWEEN,
        query = "select * from book where price between :min and :max",
        resultClass=Book.class
    )
})

public class Book {

    public static final String PRICE_BETWEEN = "Book.PriceBetween";

    // wie gehabt ...
}
```

Application

`demoPersist` persistiert vier Bücher:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10.0));
        manager.persist(new Book("2222", "Modula", 20.0));
        manager.persist(new Book("3333", "Oberon", 30.0));
        manager.persist(new Book("4444", "Eiffel", 40.0));
    });
}
```

`demoNativeQuery` zeigt die Verwendung der `EntityManager`-Methode `createNativeQuery`:

```
@SuppressWarnings("unchecked")
```

```
static void demoNativeQuery(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String sql =  
            "select * from book where PRICE between :min and :max";  
        final Query query =  
            manager.createNativeQuery(sql, Book.class);  
        query.setParameter("min", 20.0);  
        query.setParameter("max", 30.0);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

Die Ausgaben:

```
Book [2, 2222, 20.0, Modula]  
Book [3, 3333, 30.0, Oberon]
```

Die folgende Methode benutzt die `@NamedNativeQuery`-Annotationen:

```
static void demoNamedNativeQuery(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final TypedQuery<Book> query =  
            manager.createNamedQuery(Book.PRICE_BETWEEN, Book.class);  
        query.setParameter("min", 20.0);  
        query.setParameter("max", 30.0);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

Die Methode erzeugt dieselben Ausgaben wie `demoNativeQuery`.

6.14 Readonly Queries – Performance

Man kann Queries als Readonly kennzeichnen (via `Query.setHint`) – was allerdings überraschenderweise keinen wesentlichen Performance-Vorteil bringt:

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        for (int i = 0; i < 500; i++)
            manager.persist(new Book("1111" + i, "Pascal", 10));
    });
}
```

```
static void demoPerformance(TransactionTemplate tt) {
    tt.runPerfomanceTest(500,
        manager -> {
            final TypedQuery<Book> query =
                manager.createQuery("select b from Book b",
                    Book.class);
            final List<Book> books = query.getResultList();
            if (! manager.contains(books.get(0)))
                throw new RuntimeException("not expected");
        },
        manager -> {
            final TypedQuery<Book> query =
                manager.createQuery("select b from Book b",
                    Book.class);
            query.setHint(QueryHints.HINT_READONLY, true);
            final List<Book> books = query.getResultList();
            if (! manager.contains(books.get(0)))
                throw new RuntimeException("not expected");
        }
    );
}
```

Ausgaben:

```
duration[0] = 1674 milliseconds
duration[1] = 1306 milliseconds
```

7 Assoziationen

Java-Objekte stehen gewöhnlich nicht allein in der Welt herum, sondern sind per Referenzen mit anderen Objekten verbunden. Hierbei kann es sich um 1:1, 1:N oder M:N-Beziehungen handeln.

Auch Datenbank-"Objekte" stehen häufig in Verbindung miteinander. Solche Verbindungen werden realisiert mittels Primär- und Fremdschlüssel und zusätzlichen Verknüpfungstabellen.

Ein Object-relationaler Mapper muss diese datenbankseitigen Schlüsselbeziehungen transformieren in referenzielle Beziehungen der Java-Objekte – und umgekehrt.

1:1-Beziehungen sind relativ selten. Sie werden im folgenden dennoch ausführlich diskutiert, weil anhand solcher Beziehungen einige grundlegende Zusammenhänge geklärt werden können (insbesondere Lazy Loading, unidirektionale und bidirektionale Verbindungen und der sog. `join fetch`). Als Beispiel werden die Klassen `Book` und `Content` (der Inhalt eines Buches) benutzt.

Dann werden 1:N- Beziehungen diskutiert. Die N-Seite einer solchen Verbindung wird in Java mittels `Collections` (`Sets`, `Lists` etc.) implementiert. Als Beispiel dienen die Klassen `Book` und `Publisher`: Ein `Publisher` verlegt viele `Books`, ein `Book` gehört immer genau zu einem `Publisher`.

M:N-Beziehungen erfordern auf der Datenbankseite eine Verknüpfungstabelle. Auf der Java-Seite ist diese Tabelle allerdings nicht sichtbar. Beide Seiten der Verbindung werden einfach durch `Collections` implementiert. Als Beispiel dient die Beziehung `Book` und `Author`: ein `Book` kann mehrere `Authors` haben, ein `Author` kann mehrere `Books` geschrieben haben.

Schließlich werden rekursive Beziehungen diskutiert. Als Beispiel dient eine Klasse `Topic`: ein `Topic`-Objekt kann beliebig viele Kinder besitzen, welche ihrerseits wieder `Topics` sind.

7.1 one-to-one

Im folgenden werden Entities in eine einfache 1:1-Beziehung gesetzt. Ein `Book` besitzt (optional) einen `Content` (ein solcher `Content` besteht nur aus einem einfachen Text). Diese Beziehung wird hier zunächst unidirektional implementiert - und zwar so, dass ein `Book`-Objekt eine Referenz auf einen `Content` haben kann. Man kann also von einem `Book` zu seinem `Content` navigieren, aber (noch!) nicht umgekehrt.

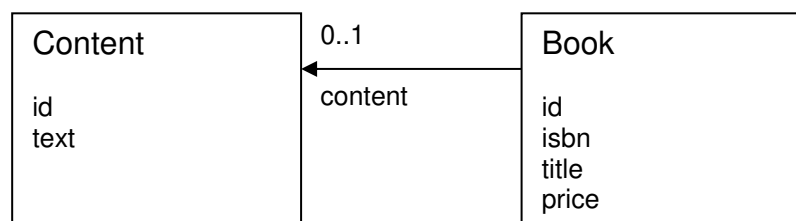
In der Datenbank benötigt die `Book`-Tabelle einen Foreign-Key, der die `Content`-Tabelle referenziert. Die Fremdschlüsselspalte wird als `CONTENT_ID` bezeichnet.

(Natürlich hätte man diese Beziehung auch umkehren können: man könnte in der `CONTENT`-Tabelle einen Fremdschlüssel `BOOK_ID` implementieren, der die `BOOK`-Tabelle referenziert... Aber diese Abbildung ist offenbar nicht die "natürliche" - man möchte i.d.R. den `CONTENT` eines `BOOKs` ermitteln - aber nicht umgekehrt das `BOOK` eines `CONTENTs`.)

Man beachte, dass in den Java-`Book`-Objekten von diesem Foreign-Key keine Rede ist. Dort gibt's nur eine Referenz vom Typ `Content` (namens `content`) - und natürlich das entsprechende setter/getter Paar: `getContent` und `setContent`.

Dieses `content`-Attribut wird - statt mit einer `@Basic` - mit einer `@OneToOne`-Annotation ausgestattet.

Klassendiagramm



create.sql

```
create table CONTENT (
    ID integer generated by default as identity (start with 1),
    TEXT varchar (1024) not null,
    primary key (ID)
);

create table BOOK (
    ID integer generated by default as identity (start with 1),
```

```
ISBN varchar (20) not null,  
TITLE varchar (128) not null,  
PRICE double not null,  
CONTENT_ID integer,  
primary key (ID),  
unique (ISBN),  
unique (CONTENT_ID),  
foreign key (CONTENT_ID) references CONTENT  
);
```

Man beachte die Klausel: `unique (CONTENT_ID)!`

Die Klasse `Content` ist trivial:

Content

```
package domain;  
// ...  
@Entity  
public class Content {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    @Basic  
    private String text;  
  
    // Konstruktoren, getter, setter, toString...  
}
```

Die Klasse `Book` ist interessanter:

Book

```
package domain;  
// ...  
@Entity  
public class Book {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    @Basic  
    private String isbn;  
  
    @Basic  
    private String title;  
  
    @Basic  
    private double price;  
}
```

```

@OneToOne
// @OneToOne(fetch=FetchType.EAGER) // THIS is the default!!!
// @OneToOne(optinal=true) // THIS is the default!!!
// @JoinColumn(name="CONTENT_ID") // THIS is the default!!!!
private Content content;

// Konstruktoren, getter, setter, toString...
}

```

Application

Wir persistieren zwei Contents und zwei Books, von denen jedes mit einem der beiden Contents assoziiert ist:

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Content c1 = new Content("Pascal-Language");
        final Content c2 = new Content("Modula-Language");
        final Book b1 = new Book("1111", "Pascal", 10, c1);
        final Book b2 = new Book("2222", "Modula", 20, c2);
        manager.persist(c1);
        manager.persist(c2);
        manager.persist(b1);
        manager.persist(b2);
    });
}

```

Man beachte, dass in `demoPersist` zunächst die Content-Objekte und dann erst die Book-Objekte an `persist` übergeben werden.

Die Ausgaben:

```

Hibernate: insert into Content (...) values (default, ?)
Hibernate: insert into Content (...) values (default, ?)
Hibernate: insert into Book (...) values (default, ?, ?, ?, ?)
Hibernate: insert into Book (...) values (default, ?, ?, ?, ?)

```

Die Datenbank:

```

CONTENT
ID TEXT
-----
1  Pascal-Language
2  Modula-Language
-----

BOOK
ID ISBN TITLE  PRICE CONTENT_ID
-----
1  1111 Pascal  10.0    1
2  2222 Modula  20.0    2

```

Zwei Query-Methoden:

```
static void demoQuerySingleResult(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b where b.isbn = :isbn";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.setParameter("isbn", "1111");
        final Book b = query.getSingleResult();
        System.out.println(b);
        System.out.println("\t" + b.getContent());
    });
}
```

Die Ausgaben:

```
Hibernate: select ... from Book where isbn=?
Hibernate: select ... from Content where id=?
Book [1, 1111, 10.0, Pascal]
    Content [1, Pascal-Language]
```

```
static void demoQueryResultList(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select b from Book b";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        final List<Book> books = query.getResultList();
        for (final Book b : books) {
            System.out.println(b);
            System.out.println("\t" + b.getContent());
        }
    });
}
```

```
Hibernate: select ... from Book where isbn=?
Hibernate: select ... from Content where id=?
Hibernate: select ... from Content where id=?
Book [1, 1111, 10.0, Pascal]
    Content [1, Pascal-Language]
Book [2, 2222, 20.0, Modula]
    Content [2, Modula-Language]
```

Man beachte, dass in den Query-Methoden nicht nur das jeweilige `Book` ausgegeben wird, sondern zugleich auch dessen `Content` - obwohl in der JPQL-Anfrage nur die `Book`-Klasse angesprochen wird.

Enthält die `BOOK`-Tabelle n Zeilen und hat jedes `BOOK` seinen eigenen `CONTENT`, so werden also $1+n$ Lesezugriffe ausgeführt. Man erkennt, dass man mit JPA offenbar einiges verkehrt machen kann (wie man es besser machen kann, wird später gezeigt...).

7.2 one-to-one : cascade

Im letzten Beispiel musste die Anwendung zunächst die `Content`-Objekte speichern, um dann die `Book`-Objekte speichern zu können. Aber die `Book`-Objekte besitzen doch jeweils eine Referenz auf ein `Content`-Objekt. Dann müsste es eigentlich reichen, nur die `Book`-Objekte an `persist` zu übergeben. JPA muss dann nur wissen, dass das Speichern eines `Book`-Objekts eine kaskadierende Wirkung haben soll - dass gleichzeitig mit einem `Book`-Objekt das mit diesem Objekt assoziierte `Content`-Objekt persistiert werden soll. Dies geschieht über ein `cascade`-Attribut in der `@OneToOne`-Annotation:

Book

```
package domain;
// ...
@Entity
public class Book {

    // wie gehabt...

    @OneToOne(cascade = { CascadeType.ALL })
    private Content content;

    // wie gehabt...
}
```

Dann müssen nunmehr die `Book`-Objekte explizit persistiert werden (die mit diesen Objekten verbundenen `Content`-Objekte werden nun implizit persistiert):

Application

Wir erzeugen zwei `Books`, wobei jedes mit einem `Content` verbunden wird. Wir rufen nur zweimal `persist` auf – und übergeben jeweils ein `Book`:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10,
            new Content("Pascal-Language")));
        manager.persist(new Book("2222", "Modula", 20,
            new Content("Modula-Language")));
    });
}
```

Die Ausgaben:

```
Hibernate: insert into Content (...) values (default, ?)
Hibernate: insert into Book (...) values (default, ?, ?, ?, ?)
Hibernate: insert into Content (...) values (default, ?)
```


Hibernate: insert into Book (...) values (default, ?, ?, ?, ?)

Jeder Aufruf der `persist`-Methode resultiert nun in zwei INSERTs: zunächst wird eine CONTENT-Zeile geschrieben, dann eine BOOK-Zeile.

Eine Query-Methode:

```
static void demoQuery(TransactionalTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query = manager.createQuery(
            "select b from Book b", Book.class);
        final List<Book> books = query.getResultList();
        for (final Book b : books) {
            System.out.println(b);
            System.out.println("\t" + b.getContent());
        }
    });
}
```

Die Ausgaben:

```
Hibernate: select ... from Book where isbn=?
Hibernate: select ... from Content where id=?
Hibernate: select ... from Content where id=?
Book [1, 1111, 10.0, Pascal]
    Content [1, Pascal-Language]
Book [2, 2222, 20.0, Modula]
    Content [2, Modula-Language]
```

7.3 one-to-one : lazy

Bislang erzeugte der JPA-Provider immer dann, wenn er ein neues `Book` produzierte, zugleich auch das zu diesem `Book` gehörige `Content`-Objekt - auch dann, wenn die Anwendung an diesem `Content`-Objekt gar nicht interessiert war. Besser wäre es, wenn auf die `CONTENT`-Tabelle nur dann zugegriffen würde, wenn die Anwendung auch tatsächlich auf das mit dem jeweiligen `Book` assoziierte `Content`-Objekt zugreifen würde. In diesem Falle spricht man dann von Lazy-Loading. Das Gegenteil von `LAZY` ist `EAGER` - wie man hat sehen können, ist `EAGER` bei 1:1-Beziehungen offensichtlich der Default. Und dieser Default muss außer Kraft gesetzt werden. Dies geschieht über das `fetch`-Attribut der `@OneToOne`-Annotation:

Book

```
package domain;
// ...
@Entity
public class Book {
    // ...
    @OneToOne(fetch=FetchType.LAZY,
        cascade = { CascadeType.ALL }, optional = true)
    private Content content;
    // ...
}
```

Application

Wir persistieren zwei mit einem `Content` assoziierte Books:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10,
            new Content("Pascal-Language")));
        manager.persist(new Book("2222", "Modula", 20,
            new Content("Modula-Language")));
    });
}
```

Die Query-Transaktionen beinhalten nun eine Reihe von Trace-Anweisungen. Der Sinn dieser Anweisungen wird bei der Analyse der Ausgaben deutlich:

```
static void demoQuerySingleResult(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query = manager.createQuery(
            "select b from Book b where b.isbn = :isbn",
            Book.class);
        query.setParameter("isbn", "1111");
        final Book book = query.getSingleResult();
        System.out.println(book);
    });
}
```

```

        System.out.println("vor b.getContent");
        final Content c = book.getContent();
        if (c != null) {
            System.out.println(Members.toString(c.getClass()));
            System.out.println("vor c.getId()");
            final Integer id = c.getId();
            System.out.println("vor c.getText()");
            final String text = c.getText();
            System.out.println("\t" + id + " " + text);
        }
    });
}

```

Die Ausgaben:

Hibernate: select ... from Book book0_ where ...

Book [1, 1111, 10.0, Pascal]

vor b.getContent

class Content_\$\$jvst299_0 extends Content

Constructors

public Content_\$\$jvst299_0()

public Content_\$\$jvst299_0(String arg0)

Fields

public static byte[] _filter_signature

private static Method[] _methods_

private MethodHandler handler

public static final long serialVersionUID

Methods

public final Object _d0clone()

public final void _d10setId(Integer arg0)

public final void _d11setText(String arg0)

public final String _d12toString()

public final boolean _d1equals(Object arg0)

public final Integer _d5getId()

public final String _d6getText()

public final int _d7hashCode()

protected final Object clone()

@Overrides public final boolean equals(Object arg0)

public MethodHandler getHandler()

public final LazyInitializer getHibernateLazyInitializer()

@Overrides public final Integer getId()

@Overrides public final String getText()

@Overrides public final int hashCode()

public void setHandler(MethodHandler arg0)

final void setId(Integer arg0)

final void setText(String arg0)

@Overrides public final String toString()

public final Object writeReplace()

vor c.getId()

Hibernate: select ... from Content content0_ where ...

vor c.getText()

1 Pascal-Language

Der Aufruf von `Query.getSingleResult` liefert aufgrund eines `SELECTs` auf die `Book`-Tabelle genau ein einziges `Book`. Zu diesem Zeitpunkt findet noch kein Zugriff auf die `CONTENT`-Tabelle statt.

Zunächst wird das `Book` ausgegeben. Dann wird auf jedes `Book` die `getContent`-Methode aufgerufen. Auch hier findet offensichtlich noch kein `SELECT` auf `CONTENT` statt. `getContent` liefert offenbar eine nicht-null-Referenz.

Mittels der Utility-Methode `Members.toString` wird die Klasse desjenigen Objekts ausgegeben, auf das die mittels `getContent` ermittelte Referenz verweist. Wie man sieht, handelt es sich nicht(!) um ein Objekt vom Typ `Content`, sondern um ein Objekt einer Klasse, die von `Content` abgeleitet ist (und die zur Laufzeit automatisch erzeugt wird). Wie nennen diese Klasse in der Folge `$$Content`. Diese Klasse überschreibt alle in `Content` definierten Methoden.

Denn wird auf die von `getContent` zurückgelieferte Referenz die Methode `getId` aufgerufen. Bevor diese zurückkehrt (also im Kontext des Aufrufs dieser Methode), wird ein Einzelsatz-`SELECT` auf `CONTENT` abgesetzt.

Wie kann nun aber im Kontext der Methode `getId` (resp. `getText`) der Zugriff auf `CONTENT` stattfinden? Man beachte, dass die `getId` der `Content`-Klasse nur aus einer einzigen Zeile besteht: `return this.id`!

Bevor wir die Funktionsweise dieses Lazy-Loading näher analysieren, sei eine weitere demo-Methode vorgestellt. Sie selektiert alle `Books` und gibt dann diese `Books` zusammen mit dem jeweiligen `Content` aus:

```
static void demoQueryResultList(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query = manager.createQuery(
            "select b from Book b",
            Book.class);
        final List<Book> books = query.getResultList();
        for (final Book b : books) {
            System.out.println(b);
            System.out.println("vor b.getContent()");
            final Content c = b.getContent();
            if (c != null) {
                System.out.println("vor c.getId()");
                final Integer id = c.getId();
                System.out.println("vor c.getText()");
                final String text = c.getText();
                System.out.println("\t" + id + " " + text);
            }
        }
    });
}
```

Die Ausgaben:

```

Hibernate: select ... from Book book0_
Book [1, 1111, 10.0, Pascal]
vor b.getContent
Hibernate: select ... from Content content0_ where content0_.id=?
    1 Pascal-Language
Book [2, 2222, 20.0, Modula]
vor b.getContent
Hibernate: select ... from Content content0_ where content0_.id=?
    2 Modula-Language

```

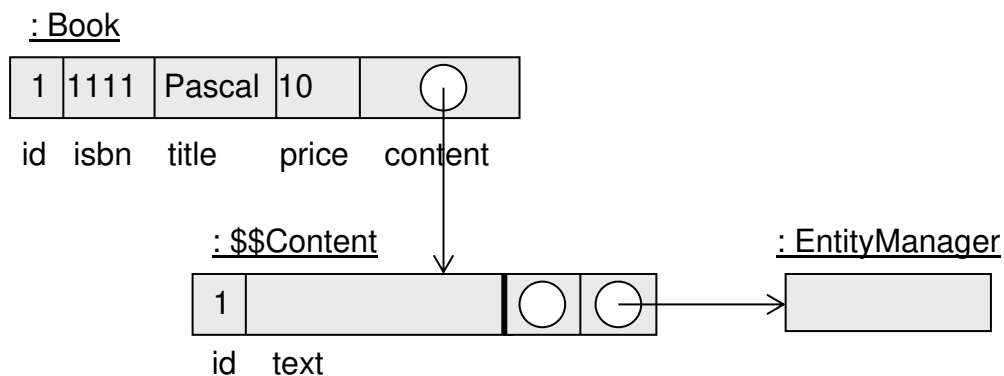
Man beachte: hier wird zweimal nachgeladen (auch das ist natürlich nicht performant...)

Man beachte weiterhin, dass das Lazy Loading natürlich nur im Kontext einer existierenden Transaktion / eines existierenden `EntityManager`s funktioniert. Nach Verlassen dieses Kontexts führt der Versuch eines Lazy Loadings zu eine `LazyInitializationException`!!!

Funktionsweise des Lazy-Loading

Die folgenden Erklärungen zum Thema Lazy-Loading sind spezifisch für Hibernate. Andere JPA-Provider werden das zugrundeliegende Problem möglicherweise anders lösen.

Immer dann, wenn Hibernate aus einer `BOOK`-Zeile ein `Book`-Objekt erzeugt, wird zugleich ein Objekt einer von `Content` abgeleiteten Klasse erzeugt (`$$Content`). Das `id`-Attribut dieses Objekts kann bereits auf `CONTENT_ID`-Spalte der gelesenen `BOOK`-Zeile gesetzt werden - alle anderen Attribute (hier: das `text`-Attribut) des Objekts sind aber nicht initialisiert worden:



Die hier als `$$Content` bezeichnete Klasse ist zur Laufzeit mittels eines Bytecode-Generation-Tools erzeugt worden. Sie existiert nur als Bytecode im Hauptspeicher. Man kann sich die Implementierung der Klasse etwa wie folgt vorstellen (eine sehr vereinfachte Sicht):

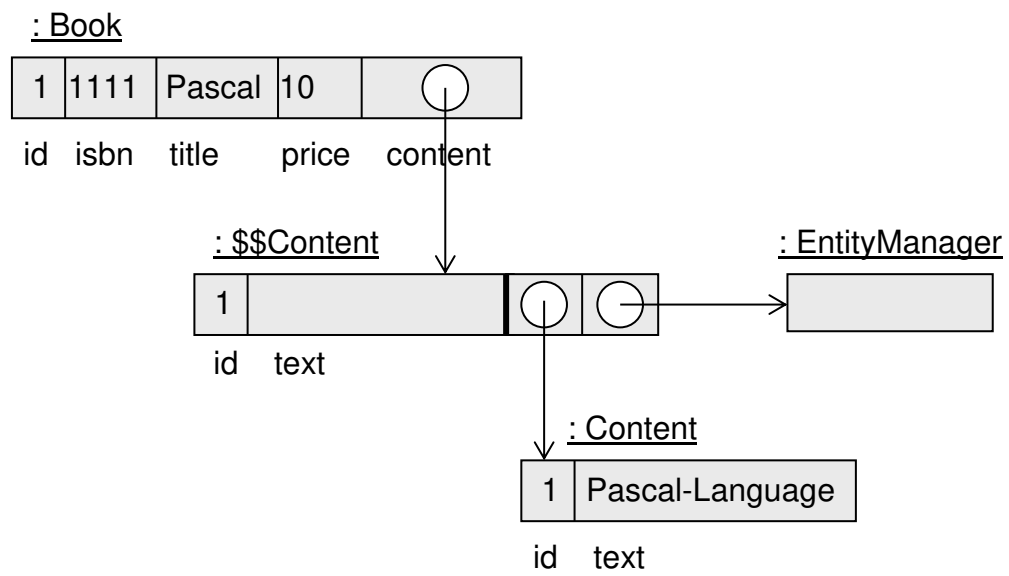
```
public class $$Content extends Content {  
  
    private EntityManager manager;  
    private Content realContent;  
  
    public Integer getId() {  
        if (this.realContent == null)  
            this.load();  
        return this.realContent.getId();  
    }  
  
    public String getText() {  
        if (this.realContent == null)  
            this.load();  
        return this.realContent.getText();  
    }  
  
    private void load () {  
        this.realContent = new Content();  
        führe unter Zuhilfenahme der Session einen SELECT  
        auf CONTENT aus, um die Felder des $$realContent  
        zu initialisieren  
    }  
}
```

Wenn nun im oben dargestellten Zustand z.B. eine der getter-Methoden aufgerufen wird, wird ein "richtiges" `Content`-Objekt erzeugt und initialisiert (dies erfordert einen Einzelsatz-`SELECT`). Dieses "richtige" Objekt wird mit dem `$$Content`-Objekt verbunden.

Diese "Laden auf Verlangen" wird als Lazy-Loading bezeichnet.

Um das Lazy-Loading zu erzwingen, muss also nur eine der getter-Methoden aufgerufen werden.

Der anschließende Zustand kann dann wie folgt dargestellt werden:



7.4 one-to-one : join

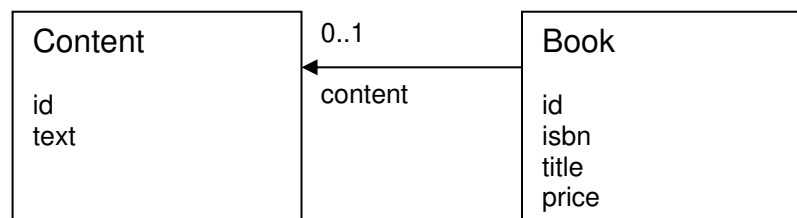
Die folgende Lösung basiert wieder auf einer unidirektionalen Beziehung: ein `Book` kennt seinen `Content` - aber nicht umgekehrt.

Und auch hier wird die `@OneToOne`-Kennzeichnung des `content`-Attributs der `Book`-Klasse wieder mit `fetch=FetchType.LAZY` versehen.

Die hier vorgestellte Lösung deklariert(!) zwar Lazy-Loading, sie vermeidet(!) aber die tatsächliche Nutzung Lazy Loading.

Die Lösung geht davon aus, dass man bei einem Query i.d.R. vorher weiß, was man mit dem Ergebnis dieses Queries anstellen möchte. Weiß man z.B., dass man nicht nur die Daten der `Books` benötigt, sondern zugleich auch zu jedem `Book` dessen `Content`, dann kann man sofort einen `join fetch` absetzen lassen - und somit mit einem einzigen Zugriff (der von der Datenbank performant ausgeführt werden wird) alle Daten ermitteln, die später benötigt werden.

Klassendiagramm



Book

```
package domain;
// ...
@Entity
public class Book {
    // ...
    @OneToOne(fetch=FetchType.LAZY, cascade = { CascadeType.ALL })
    private Content content;
    // ...
}
```

Application

Wir persistieren wieder zwei `Books`, von denen jedes mit einem `Content` verbunden ist:

```
static void demoPersist(TransactionTemplate tt) {
```

J. Nowak, H.G. Rüschenpöhler


```

        tt.run(manager -> {
            manager.persist(new Book("1111", "Pascal", 10,
                new Content("Pascal-Language")));
            manager.persist(new Book("2222", "Modula", 20,
                new Content("Modula-Language")));
        });
    }

```

Wir selektieren alle Books zusammen mit den ihnen zugehörigen Contents:

```

static void demoQueryBooksWithContents(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b join fetch b.content";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        final List<Book> books = query.getResultList();
        for (final Book b : books) {
            System.out.println(b);
            System.out.println("\t" + b.getContent());
        }
    });
}

```

Hier wird nun ein INNER JOIN erzeugt - und als Ergebnis dieses JOINs alle erforderlichen Book- und Content-Objekte erzeugt und initialisiert werden können. Insbesondere gibt's hier dann auch keine \$\$Content-Objekte mehr. Das gewünschte Ergebnis ist also performant produziert worden - ein Nachladen von Objekten ist nicht mehr erforderlich.

Die Ausgaben:

```

Hibernate: select ... from Book b
         inner join Content c on b.content_id=c.id
Book [1, 1111, 10.0, Pascal]
         Content [1, Pascal-Language]
Book [2, 2222, 20.0, Modula]
         Content [2, Modula-Language]

```

In zweiten Query wird der "unwahrscheinliche" Fall implementiert, dass man - ausgehend von einem gegebenen Content - dessen Book ausfindig machen möchte. Natürlich ist hier dann pro Content ein weiterer Einzelsatz-SELECT erforderlich.

```

static void demoQueryContentsWithBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select c from Content c";
        final TypedQuery<Content> query =
            manager.createQuery(jpql, Content.class);
        final List<Content> contents = query.getResultList();
        for (final Content c : contents) {
            System.out.println(c);
        }
    });
}

```

```
        final String jpql2 =
            "select b from Book b where b.content = :content";
        final Book b = manager.createQuery(jpql2, Book.class)
            .setParameter("content", c).getSingleResult();
        System.out.println("\t" + b);
    }
});
}
```

Die Ausgaben:

Hibernate: select ... from Content

Content [1, Pascal-Language]

Hibernate: select ... from Book b where b.content_id=?

Book [1, 1111, 10.0, Pascal]

Content [2, Modula-Language]

Hibernate: select ... from Book b where b.content_id=?

Book [2, 2222, 20.0, Modula]

7.5 one-to-one : join-fetch Performance

Im folgenden wird die Performance des Lazy-Loadings mit der Performance des join-fetch verglichen:

```
package domain;
// ...
@Entity
public class Book {
    // ...
    @OneToOne(fetch=FetchType.LAZY)
    private Content content;
    // ...
}
```

In `demoPersist` werden 500 Books persistiert (jedes Book hat seinen eigenen Content):

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        for (int i = 0; i < 500; i++) {
            final Content content = new Content("Pascal-Language" + i);
            manager.persist(content);
            final Book book = new Book(
                "1111" + i, "Pascal" + i, 10 + i, content);
            manager.persist(book);
        }
    });
}
```

In `demoPerformance` werden die 500 Books jeweils 100 mal gelesen:

```
static void demoPerformance(TransactionTemplate tt) {
    tt.runPerformanceTest(100,
        manager -> {
            final String jpql =
                "select b from Book b";
            final TypedQuery<Book> query =
                manager.createQuery(jpql, Book.class);
            final List<Book> books = query.getResultList();
            books.forEach(b -> b.getContent().getText());
        },
        manager -> {
            final String jpql =
                "select b from Book b join fetch b.content";
            final TypedQuery<Book> query =
                manager.createQuery(jpql, Book.class);
            final List<Book> books = query.getResultList();
            books.forEach(b -> b.getContent().getText());
        });
}
```

Die Ausgaben zeigen, dass die zweite `select`-Variante etwa um den Faktor 50 performanter ist (sofern Derby verwendet wird):

```
duration[0] = 4072 milliseconds  
duration[1] = 99 milliseconds
```

Sofern einem embedded-Derby-Datenbank verwendet wird, beträgt der Faktor allerdings nur etwa 5 – aber immerhin:

```
duration[0] = 3072 milliseconds  
duration[1] = 645 milliseconds
```

7.6 one-to-one : update / delete

Im folgenden geht's um `UPDATEs` und `DELETEs`. Wie im letzten Abschnitt liegt auch hier eine unidirektionale Implementierung von `Book` und `Content` zugrunde:

Book

```
package domain;
// ...
@Entity
public class Book {
    // ...
    @OneToOne(fetch=FetchType.LAZY, cascade = { CascadeType.ALL })
    private Content content;
    // ...
}
```

Application

Wir persistieren wieder zwei Books mit Contents:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10,
            new Content("Pascal-Language")));
        manager.persist(new Book("2222", "Modula", 20,
            new Content("Modula-Language")));
    });
}
```

Wir lesen das erste der beiden Books (mit der ISBN 1111) ein. Wir löschen den Content dieses Books, erzeugen einen neuen Content und verknüpfen nun diesen neuen Content mit dem Book. Der neue Content wird automatisch persistiert werden (siehe das `cascade`-Attribut des `content`-Attributs von `Book`).

```
static void demoUpdate(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b join fetch b.content " +
            "where b.isbn = :isbn";
        final Book b = manager.createQuery(
            jpql, Book.class).setParameter("isbn", "1111")
            .getSingleResult();
        manager.remove(b.getContent());
        b.setContent(new Content("PASCAL-LANGUAGE"));
    });
}
```

Die Ausgaben:

```

Hibernate: select ... from Book b
         inner join Content c on b.content_id=c.id
         where book0_.isbn=?
Hibernate: insert into Content (id, text) values (default, ?)
Hibernate: values identity_val_local()
Hibernate: update Book ... where id=?
Hibernate: delete from Content where id=?

```

In `demoDelete` ermitteln wir das zweite der beiden Books (2222) und löschen das Book. Mit dem Book wird zugleich auch der entsprechende Eintrag in der CONTENT-Tabelle gelöscht:

```

static void demoDelete(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b join fetch b.content " +
            "where b.isbn = :isbn";
        final Book b = manager.createQuery(
            jpql, Book.class).setParameter("isbn", "2222")
            .getSingleResult();
        manager.remove(b);
    });
}

```

```

Hibernate: select ...      from Book b
         inner join Content c on b.content_id=c.id
         where b.isbn=?
Hibernate: delete from Book where id=?
Hibernate: delete from Content where id=?

```

Die Datenbank nach Ausführung aller `demo`-Methoden:

```

CONTENT
ID TEXT
-----
3  PASCAL-LANGUAGE
-----

```

```

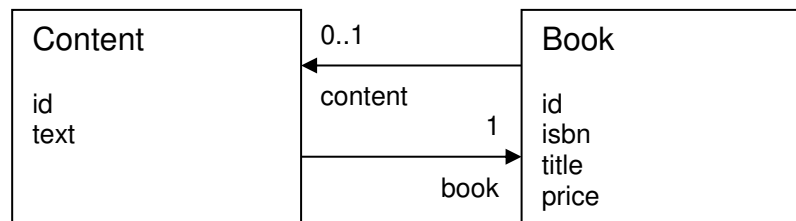
BOOK
ID ISBN TITLE  PRICE CONTENT_ID
-----
1  1111 Pascal  10.0   3
-----

```

7.7 one-to-one : bidirectional

Bislang konnte man von einem `Book` zu seinem `Content` navigieren, gelangte aber vom `Content` nicht zu dessen `Book` zurück. Im folgenden wird die bislang unidirektionale Assoziation ersetzt durch eine bidirektionale.

Klassendiagramm



create.sql

Die Schemadefinitionen bleiben unverändert. Es gibt auch hier nur einen einzigen Fremdschlüssel: die Spalte `CONTENT_ID` der Tabelle `BOOK`. (Die Datenbank enthält immer nur "unidirektionale" Beziehungen!)

Book

```

package domain;
// ...
@Entity
public class Book {

    // ...

    @OneToOne(cascade = { CascadeType.ALL }, optional = true)
    //@JoinColumn(unique=true)
    private Content content;

    Book() { }

    public Book(String isbn, String title,
                double price, Content content) {
        // ...
        this.content = content;
        if (this.content != null)
            this.content.setBook(this);
    }

    public Book(String isbn, String title, double price) {
        this(isbn, title, price, null);
    }
}
  
```

```

    }

    // ...
}

```

Man beachte, dass im Konstruktor beide Verbindungen gesetzt werden:

```

    this.content = content;
    if (this.content != null)
        this.content.setBook (this);

```

Content

```

package domain;
// ...
@Entity
public class Content {
    // ...
    @OneToOne(mappedBy = "content")
    private Book book;
    // ...
}

```

Das `book`-Attribut ist hier ebenso wie das `content`-Attribut der `Book`-Klasse mit `@OneToOne` gekennzeichnet. Diese Annotation ist hier aber mit dem Attribut `mappedBy` ausgestattet: der Wert dieses Attributs ist der Name des "inversen" Attributs auf der `Book`-Seite. Dieser `mappedBy`-Eintrag ist unbedingt erforderlich (wenngleich er nach der "Syntax" der `@OneToOne`-Annotation optional ist - der Compiler also diesen Eintrag nicht garantieren kann).

Im der folgenden Testapplikation werden beide Wege gezeigt: derjenige von den `Books` zum jeweiligen `Content` und derjenige von den `Contents` zu dem jeweiligen `Book`:

Application

Die Persist-Transaktion:

```

static void demoPersist(TransactionTemplate tt) {
    Util.mlog();
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10,
            new Content("Pascal-Language")));
        manager.persist(new Book("2222", "Modula", 20,
            new Content("Modula-Language")));
    });
}

```

Wir selektieren alle `Books` und geben diese zusammen mit ihrem jeweiligen `Content` aus:


```

static void demoQueryBooksWithContents(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Book> query = manager.createQuery(
            "select b from Book b join fetch b.content",
            Book.class);
        final List<Book> books = query.getResultList();
        for (final Book b : books) {
            System.out.println(b);
            System.out.println("\t" + b.getContent());
        }
    });
}

```

Die Ausgaben:

```

Hibernate: select ... from Book b
         inner join Content c on b.content_id=c.id
Hibernate: select ... from Book b
         left outer join Content c on b.content_id=c.id
         where b.content_id=?
Hibernate: select from Book b
         left outer join Content c on b.content_id=c.id
         where b.content_id=?
Book [1, 1111, 10.0, Pascal]
         Content [1, Pascal-Language]
Book [2, 2222, 20.0, Modula]
         Content [2, Modula-Language]

```

Wir selektieren alle **Contents** und geben diese zusammen mit ihrem jeweiligen **Book** aus:

```

static void demoQueryContentsWithBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final TypedQuery<Content> query = manager.createQuery(
            "select c from Content c join fetch c.book",
            Content.class);
        final List<Content> contents = query.getResultList();
        for (final Content c : contents) {
            System.out.println(c);
            System.out.println("\t" + c.getBook());
        }
    });
}

```

Die Ausgaben:

```

Hibernate: select ... from Content c
         inner join Book b on c.id=b.content_id
Content [1, Pascal-Language]
         Book [1, 1111, 10.0, Pascal]
Content [2, Modula-Language]
         Book [2, 2222, 20.0, Modula]

```

Man erkennt, dass Hibernate offenbar einige Mühe hat, die bidirektionalen Beziehungen aufzubauen (zumindest was den ersten Query angeht)...

Und kann sich dann die Frage stellen, ob man Hibernate diese Mühe nicht besser ersparen sollte:

Man sollte sich fragen, ob hier eine bidirektionale Beziehung überhaupt sinnvoll ist. In einem Kontext (z.B. "Bibliothek", "Online-Shop"), in dem es um Bücher und Inhalte geht, wird man natürlich folgende Frage häufig beantworten müssen: Welchen Inhalt hat dieses oder jenes Buch? Aber die umgekehrte Frage: zu welchem Buch gehört ein (gegebener) Inhalt? ist sicher nicht allzu wahrscheinlich. Also ist es sinnvoll, von einem `Book` zu dessen `Content` navigieren zu können - der umgekehrte Weg aber muss nicht unbedingt unterstützt werden.

Und unidirektionale Beziehungen sind nicht nur performanter, sondern natürlich auch einfacher zu implementieren. Sollte es dann doch einmal erforderlich sein, von einem gegebenen `Content` zu dessen `Book` zu gelangen, kann einfach eine zusätzliche Query abgesetzt werden (s. nächster Abschnitt).

7.8 many-to-one

Im Folgenden werden Entities in eine einfache N:1-Beziehung gesetzt. Ein `Book` wird bei einem `Publisher` verlegt. Einem `Publisher` können `n` `Books` zugeordnet werden. Diese Beziehung wird hier zunächst unidirektional implementiert - und zwar so, dass ein `Book`-Objekt eine Referenz auf den `Publisher` besitzt. Man kann also von einem `Book` zu einem `Publisher` navigieren, aber nicht umgekehrt.

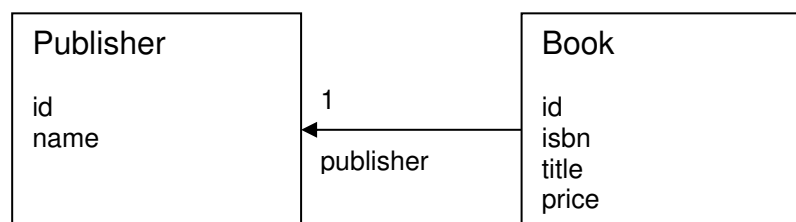
In der Datenbank benötigt die `BOOK`-Tabelle einen Foreign-Key, der die `PUBLISHER`-Tabelle referenziert. Die Fremdschlüsselspalte wird (wir orientieren uns auch hier wieder am default-Namen) als `PUBLISHER_ID` bezeichnet.

create.sql

```
create table PUBLISHER (  
    ID integer generated by default as identity (start with 1),  
    NAME varchar (128) not null,  
    primary key (ID),  
    unique (NAME)  
);  
  
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    ISBN varchar (20) not null,  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    PUBLISHER_ID integer,  
    primary key (ID),  
    unique (ISBN),  
    foreign key (PUBLISHER_ID) references PUBLISHER  
);
```

Man beachte, dass auch hier in den Java-`Book`-Objekten von diesem Foreign-Key keine Rede ist. Dort gibt's nur ein Attribut vom Typ `Publisher` (namens `publisher`) - und natürlich das entsprechende setter/getter Paar: `getPublisher` und `setPublisher`.

Klassendiagramm



Publisher

```
package domain;
// ...
@Entity
public class Publisher {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;
}
```

Book

```
package domain;
// ...
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    @ManyToOne(fetch=FetchType.LAZY)
    // @JoinColumn(name="PUBLISHER_ID") // THIS is the default!!!!
    private Publisher publisher;

    // Konstruktoren, getter, setter, toString...
}
```

"Viele" Book-Objekte beziehen sich auf jeweils "ein" Objekt der Gegenseite - auf ein Publisher-Objekt - daher @ManyToOne.

Man beachte, dass bei @ManyToOne kein cascade-Attribut angegeben ist (obwohl auch hier die Angabe von cascade möglich ist). fetch ist wieder auf LAZY gesetzt (man möchte also nicht automatisch bei einem Book auch dessen Publisher bekommen!).

Application

Wie persistieren zwei `Publisher` und vier `Books`. Die ersten drei `Books` gehören zum ersten `Publisher`, das letzte `Book` zum zweiten `Publisher`:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 30, p1));
        manager.persist(new Book("2222", "Modula", 40, p1));
        manager.persist(new Book("3333", "Oberon", 50, p1));
        manager.persist(new Book("4444", "Eiffel", 20, p2));
    });
}
```

Wir ermitteln alle `Books` und jeweils den `Publisher` des jeweiligen `Books`:

```
static void demoQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b join fetch b.publisher";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.getResultList().forEach(b -> {
            System.out.println(b);
            System.out.println("\t" + b.getPublisher());
        });
    });
}
```

Die Ausgaben:

```
Hibernate: select ... from Book b
         inner join Publisher p on b.publisher_id=p.id
Book [1, 1111, 30.0, Pascal]
  Publisher [1, Addison]
Book [2, 2222, 40.0, Modula]
  Publisher [1, Addison]
Book [3, 3333, 50.0, Oberon]
  Publisher [1, Addison]
Book [4, 4444, 20.0, Eiffel]
  Publisher [2, Prentice]
```

Zwar kann man nicht von einem `Publisher` zu seinen `Books` navigieren – aber man kann diese `Books` natürlich wiederum mittels eines "inversen" Selects ermitteln (in welchem wir natürlich nicht(!) mit `join fetch` argumentieren können):

```
static void demoQueryInverse(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select p from Publisher p";
        final TypedQuery<Publisher> query =
```

```

        manager.createQuery(jpgsql, Publisher.class);
        query.getResultList().forEach(p -> {
            System.out.println(p);
            final String jpqlInverse =
                "select b from Book b where " +
                "b.publisher = :publisher";
            final TypedQuery<Book> queryInverse =
                manager.createQuery(jpqlInverse, Book.class);
            queryInverse.setParameter("publisher", p);
            final List<Book> books = queryInverse.getResultList();
            books.forEach(b -> System.out.println("\t" + b));
        });
    });
}

```

Die Ausgaben:

```

Hibernate: select ... from Publisher p
Publisher [1, Addison]
Hibernate: select ... from Book b where b.publisher_id=?
    Book [1, 1111, 30.0, Pascal]
    Book [2, 2222, 40.0, Modula]
    Book [3, 3333, 50.0, Oberon]
Publisher [2, Prentice]
Hibernate: select ... from Book b where b.publisher_id=?
    Book [4, 4444, 20.0, Eiffel]

```

Eine letzte Transaktion – wir löschen ein Book (der Publisher dieses Books bleibt natürlich erhalten):

```

static void demoRemove(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b where b.isbn = :isbn";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.setParameter("isbn", "2222");
        final Book b = query.getSingleResult();
        manager.remove(b);
    });
}

```

Der Zustand der Datenbank nach Ausführung dieser Remove-Transaktion:

```

PUBLISHER
ID NAME
-----
1  Addison
2  Prentice
-----

```

```

BOOK

```

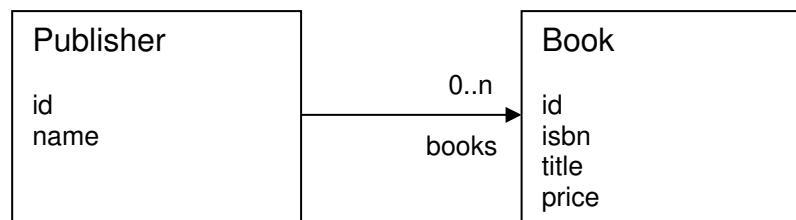
ID	ISBN	TITLE	PRICE	PUBLISHER_ID
1	1111	Pascal	30.0	1
3	3333	Oberon	50.0	1
4	4444	Eiffel	20.0	2

7.9 one-to-many

Das Datenbankschema wird aus dem letzten Abschnitt unverändert übernommen.

Statt in den `@Entity`-Klassen aber eine Beziehung von `Book` nach `Publisher` zu implementieren, wird hier die Beziehung vom `Publisher` zu seinen `Books` implementiert - eine "mehrwertige" Beziehung.

Klassendiagramm



Publisher

```

package domain;
// ...
@Entity
public class Publisher {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;

    @OneToMany(fetch=FetchType.LAZY) // LAZY is HERE the default
    @JoinColumn(name="PUBLISHER_ID") // required!!!
    private Collection<Book> books = new ArrayList<Book> ();

    // Konstruktoren, getter, setter, toString...
}
  
```

Ein `Publisher` benötigt in irgendeiner Weise eine `Collection`, in welcher Referenzen auf seine `Books` gespeichert werden können:

```
private Collection<Book> books = new ArrayList<Book> ();
```

Man beachte hier, dass die Variable `books` vom allgemeinen Interface-Typ `Collection` ist. Das Objekt, das erzeugt wird, ist vom Typ `ArrayList`.

Wichtig ist hier, dass die Variable NICHT vom Typ `ArrayList` sein darf. Sie muss ein Interface-Typ sein: entweder `Collection`, `List` oder `Set`. (Wobei im letzteren Falle dann das erzeugte Objekt natürlich z.B. vom Typ `HashSet` sein müsste.) Was für den Typ des Attributs gilt, gilt natürlich auch für den Typ der entsprechenden Property: auch dieser muss ein Interface-Typ sein. Der Grund hierfür wird später deutlich werden.

Man beachte dann die `@OneToMany`-Annotation ("ein" `Publisher` kann "viele" `Books` haben): auch hier ist wieder `LAZY` eingestellt. Dies ist hier (ANDERS als bei `@ManyToOne`!) allerdings auch der Default.

Und man beachte schließlich, dass eine explizite `@JoinColumn`-Annotation existiert. Diese Annotation ist hier (wiederum ANDERS als bei `@ManyToOne`) notwendig!

Book

```
package domain;
// ...
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    Book() { }

    public Book(String isbn, String title, double price,
        Publisher publisher) {
        // ...
        publisher.getBooks().add(this);
    }

    // getter, setter, toString...
}
```

Über den parametrisierten Konstruktor wird sichergestellt, dass ein `Book` auf jeden Fall einen `Publisher` hat: das `Book` fügt sich selbst in die `Collection<Book>` des `Publishers` ein.

Application

Wir persistieren wieder zwei Publisher und vier Books:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 30, p1));
        manager.persist(new Book("2222", "Modula", 40, p1));
        manager.persist(new Book("3333", "Oberon", 50, p1));
        manager.persist(new Book("4444", "Eiffel", 20, p2));
    });
}
```

Wir selektieren alle Publisher und wandern von den Publishern zu dessen Books:

```
static void demoQuery(TransactionTemplate tt) {
    Util.mlog();
    tt.run(manager -> {
        final String jpql =
            "select distinct p from Publisher p";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        query.getResultList().forEach(p -> {
            System.out.println(p);
            System.out.println(p.getBooks().getClass());
            p.getBooks().forEach(
                b -> System.out.println("\t" + b));
        });
    });
}
```

Die Ausgaben:

```
Hibernate: select distinct ... from Publisher p
         inner join Book b on p.id=b.PUBLISHER_ID
Publisher [1, Addison]
class org.hibernate.collection.internal.PersistentBag
Hibernate: select ... from Book b where b.PUBLISHER_ID=?
         Book [1, 1111, 30.0, Pascal]
         Book [2, 2222, 40.0, Modula]
         Book [3, 3333, 50.0, Oberon]
Publisher [2, Prentice]
class org.hibernate.collection.internal.PersistentBag
Hibernate: select ... from Book b where b.PUBLISHER_ID=?
         Book [4, 4444, 20.0, Eiffel]
```

Man beachte diejenigen Stellen, an denen Lazy Loading stattfindet.

Hier die inverse Abfrage: Wir selektieren alle Books und ermitteln zu jedem Book dessen Publisher:

```

static void demoQueryInverse(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select b from Book b";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.getResultList().forEach(b -> {
            System.out.println(b);
            final String jpqlInverse =
                "select p from Publisher p where :book " +
                "in elements (p.books)";
            final TypedQuery<Publisher> queryInverse = manager
                .createQuery(jpqlInverse, Publisher.class);
            queryInverse.setParameter("book", b);
            final Publisher p = queryInverse.getSingleResult();
            System.out.println("\t" + p);
        });
    });
}

```

Die Ausgaben:

```

Hibernate: select ... from Book
Book [1, 1111, 30.0, Pascal]
Hibernate: select ... from Publisher p where ? in (
    select b.id from Book b where p.id=b.PUBLISHER_ID)
    Publisher [1, Addison]
Book [2, 2222, 40.0, Modula]
Hibernate: select ... from Publisher p where ? in (
    select b.id from Book b where p.id=b.PUBLISHER_ID)
    Publisher [1, Addison]
Book [3, 3333, 50.0, Oberon]
Hibernate: select ... from Publisher p where ? in (
    select b.id from Book b where p.id=b.PUBLISHER_ID)
    Publisher [1, Addison]
Book [4, 4444, 20.0, Eiffel]
Hibernate: select ... from Publisher p where ? in (
    select b.id from Book b where p.id=b.PUBLISHER_ID)
    Publisher [2, Prentice]

```

In `demoRemove` wird der Publisher "Addison" gelöscht. Da `@OneToMany` kein `cascade`-Attribut besitzt, bleiben die `BOOK`-Zeilen, die der `PUBLISHER`-Zeile zuvor zugeordnet waren, natürlich erhalten:

```

static void demoRemove(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select p from Publisher p where p.name = :name";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        query.setParameter("name", "Addison");
        final Publisher p = query.getSingleResult();
    });
}

```

```

        manager.remove(p);
    });
}

```

Die Ausgaben:

```

Hibernate: select ... from Publisher where.name=?
Hibernate: update Book set PUBLISHER_ID=null where PUBLISHER_ID=?
Hibernate: delete from Publisher where id=?

```

Die Datenbank nach Ausführung aller Transaktionen:

```

PUBLISHER
ID NAME
-----
2  Prentice
-----

BOOK
ID ISBN TITLE  PRICE PUBLISHER_ID
-----
1  1111 Pascal  30.0  NULL
2  2222 Modula  40.0  NULL
3  3333 Oberon  50.0  NULL
4  4444 Eiffel  20.0  2
-----

```

Funktionsweise des Lazy-Loading

Auch hier geht's wieder darum, wie Hibernate als spezieller JPA-Provider das Lazy Loading realisiert - hier aber eben "die andere Seite" dieses Lazy-Loadings.

Angenommen, es wird ein `Publisher` gelesen:

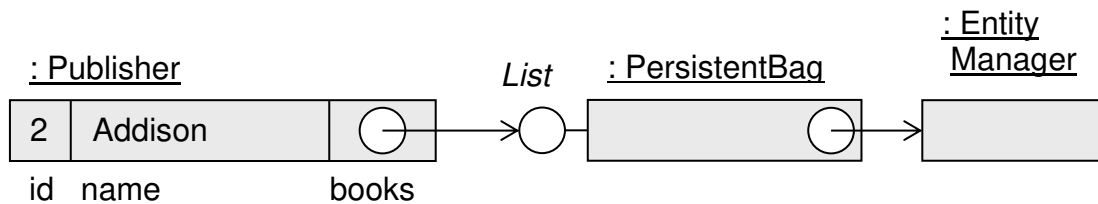
```

TypedQuery<Publisher> query = manager.createQuery(
    "select p from Publisher p where p.name = :name",
    Publisher.class);
query.setParameter("name", "Addison");
Publisher p = query.getSingleResult();

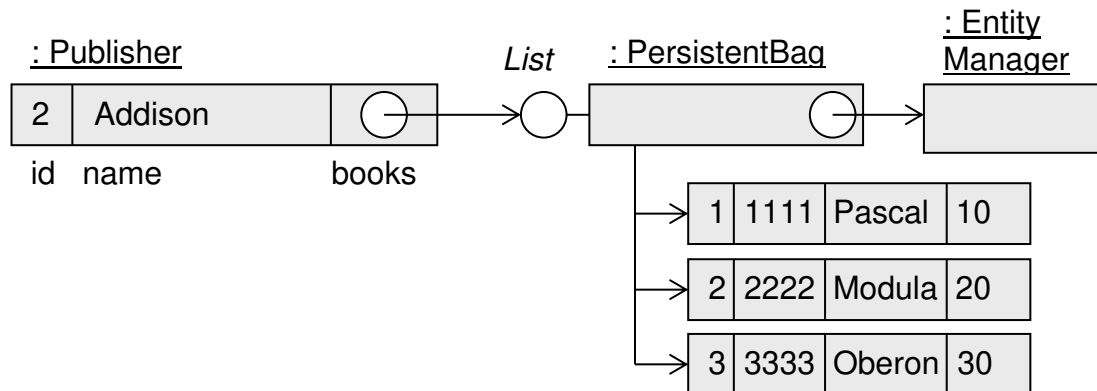
```

Dann wird natürlich ein "einfaches" `Publisher`-Objekt erzeugt (hier KEIN `$$Publisher`!) und entsprechend mit den Daten der Tabellenzeile initialisiert. Beim Erzeugen des `Publishers` ist bereits eine `ArrayList` erzeugt worden. Nach der Initialisierung bindet Hibernate nun aber sofort ein Objekt des Hibernate-Typs `PersistentBag` an das `books`-Attribut (eine Klasse, die genauso wie `ArrayList` das Interface `List` implementiert). Dieses Objekt kennt wieder den `EntityManager` (genauso wie das beim `$$Publisher` der Fall war) - es ist aber noch nicht weiter

initialisiert (enthält also noch keine `Book`-Daten). Wenn die Anwendung nun also die Referenz auf den `Publisher` erhält, sieht dieser wie folgt aus:



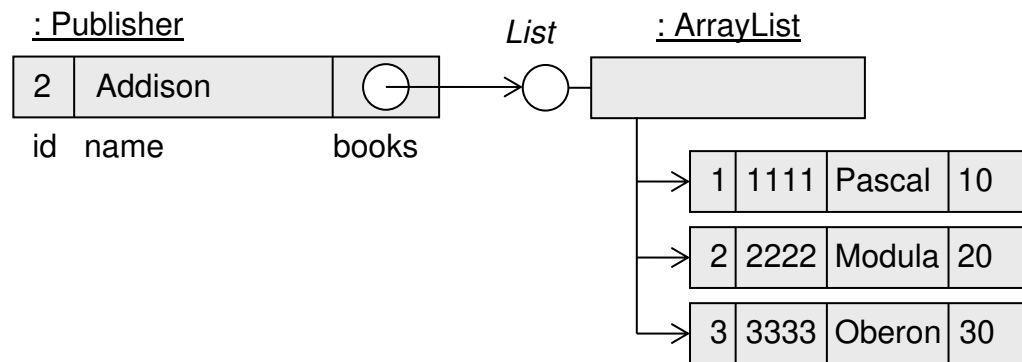
Beim obigen Aufruf von `getPublisher` hat also noch kein Zugriff auf die `BOOK`-Tabelle stattgefunden. Dieser findet erst dann statt, wenn z.B. die `size`- oder die `iterator`-Methode auf den `PersistentBag` aufgerufen wird. Da das `PersistentBag`-Objekt "intelligent" ist und den `EntityManager` kennt, können nun zum `Publisher` gehörende `BOOK`-Zeilen gelesen und entsprechende `Book`-Objekte erzeugt werden. (Das heißt natürlich auch, dass das `PersistentBag` in irgendeiner Weise bereits das auszuführende `SELECT`-Statement kennen muss...) Nachdem also das Lazy-Loading stattgefunden hat, sieht die Situation wie folgt aus:



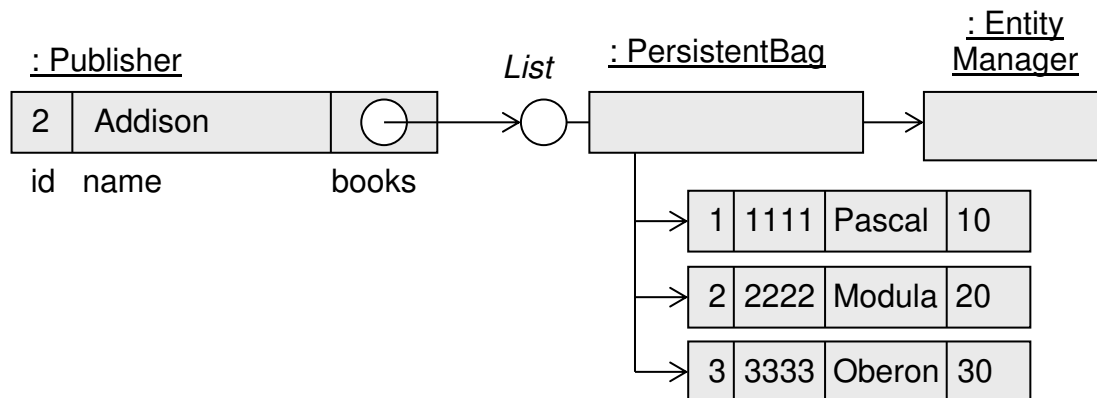
Um hier das Lazy-Loading zu erzwingen, genügt es also, irgendeine der `List`-Methoden aufzurufen (z.B. `size`, `get` oder `iterator`). Was Java5 angeht, sollte klar sein, dass die neue `for`-Schleife intern die `iterator`-Methode aufruft - also kann das Nachladen oder mittels dieser `for`-Schleife erzwungen werden.

Wird umgekehrt zunächst von der Anwendung ein transientes `Publisher`-Objekt erzeugt, um `Book`-Objekte bereichert und dann per `persist` persistiert, wird dem `Publisher` ebenfalls ein `PersistentBag`-Objekte "untergeschoben".

Vor dem `persist` sieht die Sache wie folgt aus:



Nach dem `persist` ist das `ArrayList` durch einen `PersistentBag` ersetzt worden:



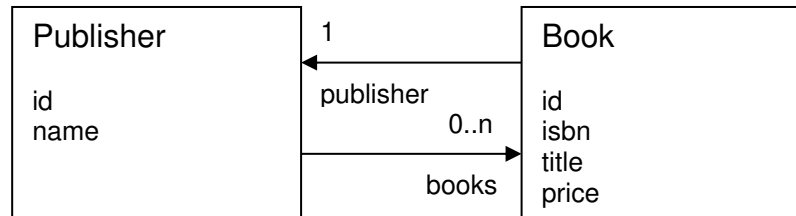
Auch in der `persist`-Methode ist also das `books`-Attribut neu gesetzt worden. Hibernate (also der aktuelle `EntityManager`) kann nun also vom `PersistentBag` über weitere Hinzufügungen oder Löschungen von `Book`-Objekten informiert werden.

(Natürlich hätten wir in der ersten Query-Methode auch wieder `join-fetch` nutzen können – so dass das Lazy Loading sich erübrigt hätte...)

7.10 one-to-many, many-to-one

Im folgenden geht's um die bidirektionale Verbindung zwischen `Book`- und `Publisher`-Objekten.

Klassendiagramm



Book

```

package domain;
// ...
@Entity
public class Book {
    // ...
    @ManyToOne(fetch=FetchType.LAZY)
    private Publisher publisher;

    Book() { }
    public Book(String isbn, String title, double price,
                Publisher publisher) {
        // ...
        this.publisher = publisher;
        publisher.getBooks().add(this);
    }
    // ...
}
  
```

Publisher

```

package domain;
// ...
@Entity
public class Publisher {
    // ...
    @OneToMany(mappedBy="publisher")
    // @JoinColumn(name="PUBLISHER_ID") // NOT required!!!
    private Collection<Book> books = new ArrayList<Book> ();
    // ...
}
  
```

Man beachte hier das `mappedBy`-Attribut der `@OneToMany`-Annotation. Hier wird das "inverse" Attribut der "anderen Seite" angegeben - das `publisher`-Attribut der `Book`-Klasse.

`mappedBy` kann NUR bei `@OneToMany` verwendet werden - NICHT bei `@ManyToOne`! Der Grund: die `@ManyToOne`-Beziehung ist diejenige Beziehung, die auch in der Datenbank "real" ist - in Form eines Foreign-Keys vorliegt. Die `@OneToMany`-Beziehung ist eine "abgeleitete", eben "inverse" Beziehung, die tatsächlich durch den Fremdschlüssel der "anderen Seite" abgebildet wird (`mappedBy`).

Die `@JoinColumn` ist hier im Gegensatz zum letzten Abschnitt wiederum nicht mehr erforderlich - die "reale" Seite der Abbildung ist ja nun in der `Book`-Klasse implementiert.

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison Wesley");
        final Publisher p2 = new Publisher("Prentice Hall");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 30, p1));
        manager.persist(new Book("2222", "Modula", 40, p1));
        manager.persist(new Book("3333", "Oberon", 50, p1));
        manager.persist(new Book("4444", "Eiffel", 20, p2));
    });
}
```

```
static void demoQueryPublishers(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select p from Publisher p";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        query.getResultList().forEach(p -> {
            System.out.println(p);
            p.getBooks().forEach(
                b -> System.out.println("\t" + b));
        });
    });
}
```

Die Ausgaben:

```
Hibernate: select ... from Publisher
Publisher [1, Addison Wesley]
Hibernate: select ... from Book where publisher_id=?
Book [1, 1111, 30.0, Pascal]
Book [2, 2222, 40.0, Modula]
Book [3, 3333, 50.0, Oberon]
Publisher [2, Prentice Hall]
```


Hibernate: select ... from Book where publisher_id=?
Book [4, 4444, 20.0, Eiffel]

```
static void demoQueryBooks(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql = "select b from Book b";  
        final TypedQuery<Book> query =  
            manager.createQuery(jpql, Book.class);  
        query.getResultList().forEach(b -> {  
            System.out.println(b);  
            System.out.println("\t" + b.getPublisher());  
        });  
    });  
}
```

Die Ausgaben:

Hibernate: select ... from Book
Book [1, 1111, 30.0, Pascal]
Hibernate: select ... from Publisher where id=?
Publisher [1, Addison Wesley]
Book [2, 2222, 40.0, Modula]
Publisher [1, Addison Wesley]
Book [3, 3333, 50.0, Oberon]
Publisher [1, Addison Wesley]
Book [4, 4444, 20.0, Eiffel]
Hibernate: select ... from Publisher where id=?
Publisher [2, Prentice Hall]

7.11 one-to-many, many-to-one : join-fetch

Im folgenden wird gezeigt, wie bei bidirektionalen 1:N-Verbindungen der `join fetch` verwendet werden kann.

Die Klassen `Book` und `Publisher` sind exakt dieselben wie im letzten Abschnitt.

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison Wesley");
        final Publisher p2 = new Publisher("Prentice Hall");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 30, p1));
        manager.persist(new Book("2222", "Modula", 40, p1));
        manager.persist(new Book("3333", "Oberon", 50, p1));
        manager.persist(new Book("4444", "Eiffel", 20, p2));
    });
}
```

```
static void demoQueryPublishers(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select distinct p from Publisher p " +
            "join fetch p.books";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        query.getResultList().forEach(p -> {
            System.out.println(p);
            p.getBooks().forEach(
                b -> System.out.println("\t" + b));
        });
    });
}
```

Die Ausgaben:

```
Hibernate: select distinct ... from Publisher p
inner join Book b on p.id=b.publisher_id
Publisher [1, Addison Wesley]
    Book [1, 1111, 30.0, Pascal]
    Book [2, 2222, 40.0, Modula]
    Book [3, 3333, 50.0, Oberon]
Publisher [2, Prentice Hall]
    Book [4, 4444, 20.0, Eiffel]
```

```
static void demoQueryBooks(TransactionTemplate tt) {
    tt.run(manager -> {
```

```
        final String jpql =
            "select b from Book b " +
            "join fetch b.publisher p";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        query.getResultList().forEach(b -> {
            System.out.println(b);
            System.out.println("\t" + b.getPublisher());
        });
    });
}
```

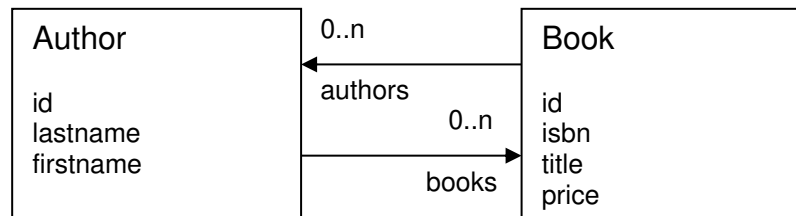
Die Ausgaben:

```
Hibernate: select ... from Book b
         inner join Publisher p on b.publisher_id=p.id
Book [1, 1111, 30.0, Pascal]
    Publisher [1, Addison Wesley]
Book [2, 2222, 40.0, Modula]
    Publisher [1, Addison Wesley]
Book [3, 3333, 50.0, Oberon]
    Publisher [1, Addison Wesley]
Book [4, 4444, 20.0, Eiffel]
    Publisher [2, Prentice Hall]
```

7.12 many-to-many

Im folgenden geht's um M:N-Beziehungen - aber nur um solche M:N-Beziehungen, welche keine zusätzlichen Assoziationsattribute besitzen.

Klassendiagramm



Bekanntlich erfordert die Abbildung solcher Beziehungen in der Datenbank eine Verknüpfungstabelle, deren Zeilen jeweils Schlüssel der beiden Basistabellen beinhalten. Die Kombination dieser Schlüssel fungiert als Primary Key für die Verknüpfungstabelle; die Schlüssel ihrerseits fungieren einzeln jeweils als Foreign Keys, welche jeweils eine Zeile die beiden Basistabellen identifizieren.

create.sql

```

create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    primary key (ID),
    unique (ISBN)
);

create table AUTHOR (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table BOOK_AUTHOR (
    BOOKS_ID integer,
    AUTHORS_ID integer,
    primary key (BOOKS_ID, AUTHORS_ID),
    foreign key (BOOKS_ID) references BOOK,
    foreign key (AUTHORS_ID) references AUTHOR
);
  
```

Man beachte die Namenswahl bei der Verknüpfungstabelle:

Der Name der Verknüpfungstabelle setzt sich zusammen aus den Namen der beiden Basistabellen (`BOOK_AUTHOR`). Dies wird per Default von JPA vorausgesetzt. (Natürlich kann man von aber auch von diesem Default abweichen...)

Die beiden Felder der Verknüpfungstabelle haben die Namen `BOOKS_ID` und `PUBLISHERS_ID` (also Pluralform der Namen der Basistabellen + `ID`). Auch dies wird von JPA als Default vorausgesetzt (auch hiervon kann man natürlich abweichen - muss dann aber diese Abweichungen in den entsprechenden Annotations explizit definieren).

Book

```
package domain;
// ...
@Entity
public class Book {
    // ...

    @ManyToMany
    // @JoinTable(name="BOOK_AUTHOR",
    //             joinColumns={@JoinColumn(name="BOOKS_ID")},
    //             inverseJoinColumns={@JoinColumn(name="AUTHORS_ID")})
    //             this is the default!!!!
    private Collection<Author> authors = new ArrayList<Author>();

    // Konstruktor, getter, setter, toString...
}
```

Man beachte die Möglichkeit, mittels einer `@JoinTable`-Annotation von den mit der Verknüpfungstabelle verbundenen Default-Namen abweichen zu können. Man beachte weiterhin, dass `@ManyToMany` per default Lazy Loading impliziert.

Author

```
package domain;
// ...
@Entity
public class Author {
    // ...

    @ManyToMany(mappedBy="authors")
    private Collection<Book> books = new ArrayList<Book>();

    // Konstruktor, getter, setter, toString...
}
```

Man beachte hier das `mappedBy`-Attribut von `@ManyToMany`.

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison Wesley");
        final Publisher p2 = new Publisher("Prentice Hall");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 30, p1));
        manager.persist(new Book("2222", "Modula", 40, p1));
        manager.persist(new Book("3333", "Oberon", 50, p1));
        manager.persist(new Book("4444", "Eiffel", 20, p2));
    });
}
```

Der Hibernate-Trace:

```
Hibernate: insert into Author (...) values (default, ?)
...
Hibernate: insert into Book (...) values (default, ?, ?, ?)
...
Hibernate: insert into Book_Author (...) values (?, ?)
...
```

```
static void demoQueryPublishers(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select distinct p from domain.Publisher p " +
            "join fetch p.books";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        query.getResultList().forEach(p -> {
            System.out.println(p);
            p.getBooks().forEach(b -> System.out.println("\t" + b));
        });
    });
}
```

Die Ausgaben:

```
Hibernate: select ... from Author a
inner join Book_Author ba on a.id=ba.authors_id
inner join Book b on ba.books_id=b.id
Author [1, Wirth]
    Book [2, 2222, 20.0, Modula]
    Book [3, 3333, 30.0, Oberon]
Author [1, Wirth]
    Book [2, 2222, 20.0, Modula]
    Book [3, 3333, 30.0, Oberon]
Author [2, Raiser]
    Book [2, 2222, 20.0, Modula]
```

Author [3, Meyer]

Book [1, 1111, 10.0, Pascal]

```
static void demoQueryBooks(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql =  
            "select b from Book b join fetch b.publisher p";  
        final TypedQuery<Book> query =  
            manager.createQuery(jpql, Book.class);  
        query.getResultList().forEach(b -> {  
            System.out.println(b);  
            System.out.println("\t" + b.getPublisher());  
        });  
    });  
}
```

Die Ausgaben:

```
Hibernate: select ... from Book b  
        inner join Book_Author ba on b.id=ba.books_id  
        inner join Author a on ba.authors_id=a.id  
Book [1, 1111, 10.0, Pascal]  
    Author [3, Meyer]  
Book [2, 2222, 20.0, Modula]  
    Author [1, Wirth]  
    Author [2, Raiser]  
Book [2, 2222, 20.0, Modula]  
    Author [1, Wirth]  
    Author [2, Raiser]  
Book [3, 3333, 30.0, Oberon]  
    Author [1, Wirth]
```

7.13 Rekursive Assoziationen

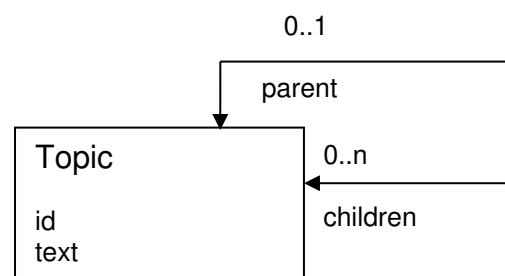
Bücher können Themen zugeordnet werden. Ein Thema (ein `Topic`) kann mehrere Unterthemen (`children`) besitzen. Ein Unterthema gehört immer genau zu einem übergeordneten Thema (`parent`). Die Beziehungen der Themen zueinander bilden also einen Baum. Jeder `Topic`-Knoten hat beliebig viele Kindknoten und genau einen Vaterknoten (bis auf den obersten Knoten: der ist vaterlos).

`Topics` könnten etwa wie folgt zu einem Baum zusammengefügt sein:

```
Alle Themen
  Fremdsprachen
    Latein
    Altgriechisch
  Informatik
    Programmiersprachen
      Pascal
      Oberon
      Modula
    Comilerbau
    Betriebssysteme
```

Dieses `Topic`-Beispiel wird im folgenden als Beispiel für rekursiv definierte Beziehungen diskutiert. Dabei wird von den Büchern der Einfachheit halber abgesehen. Aber es ist klar, dass jedem `Topic` beliebig viele `Books` zugeordnet werden können. Und einem `Book` vielleicht mehrere `Topics`... (Die Baumstruktur der `Topics` müsste dann erweitert werden um eine M:N-Beziehung zwischen `Topics` und `Books`).

Klassendiagramm



create.sql

```
create table TOPIC (
  ID integer generated by default as identity (start with 1),
  TEXT varchar (20) not null,
  PARENT_ID integer,
  primary key (ID),
  foreign key (PARENT_ID) references TOPIC
```



```
)
```

Topic

```
package domain;
// ...
@Entity
public class Topic {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String text;

    @ManyToOne
    private Topic parent;

    @OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
    private Collection<Topic> children = new ArrayList<Topic> ();

    public void add(Topic topic) {
        this.children.add(topic);
        topic.parent = this;
    }

    // Konstruktoren, getter, setter, toString...
}
```

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Topic root = new Topic("Alle Themen");
        final Topic t1 = new Topic("Informatik");
        final Topic t11 = new Topic("Comilerbau");
        final Topic t12 = new Topic("Betriebssysteme");
        final Topic t13 = new Topic("Programmiersprachen");
        final Topic t131 = new Topic("Pascal");
        final Topic t132 = new Topic("Modula");
        final Topic t133 = new Topic("Oberon");
        final Topic t2 = new Topic("Fremdsprachen");
        final Topic t21 = new Topic("Latein");
        final Topic t22 = new Topic("Altgriechisch");

        root.add(t1);
        t1.add(t11);
        t1.add(t12);
        t1.add(t13);
        t13.add(t131);
        t13.add(t132);
        t13.add(t133);
        root.add(t2);
    });
}
```

```

        t2.add(t21);
        t2.add(t22);

        manager.persist(root);
    });
}

```

```

static void demoQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select t from Topic t where t.parent is null";
        final TypedQuery<Topic> query =
            manager.createQuery(jpql, Topic.class);
        final Topic topic = query.getSingleResult();
        showTopic(topic, 0);
    });
}

```

```

public static void showTopic(Topic topic, int depth) {
    for (int i = 0; i < depth; i++)
        System.out.print("\t");
    System.out.println(topic);
    final Collection<Topic> children = topic.getChildren();
    children.forEach(child -> showTopic(child, depth + 1));
}

```

Die Ausgaben:

```

Hibernate: select ... form topic where ...
Topic [1, Alle Themen]
Hibernate: select ... form topic where ...
    Topic [2, Informatik]
Hibernate: select ... form topic where ...
        Topic [3, Comilerbau]
Hibernate: select ... form topic where ...
            Topic [4, Betriebssysteme]
Hibernate: select ... form topic where ...
                Topic [5, Programmiersprachen]
Hibernate: select ... form topic where ...
                    Topic [6, Pascal]
Hibernate: select ... form topic where ...
                        Topic [7, Modula]
Hibernate: select ... form topic where ...
                            Topic [8, Oberon]
Hibernate: select ... form topic where ...
                                Topic [9, Fremdsprachen]
Hibernate: select ... form topic where ...
                                    Topic [10, Latein]
Hibernate: select ... form topic where ...
                                        Topic [11, Altgriechisch]
Hibernate: select ... form topic where ...

```

Die Datenbank nach Ausführung des Programms:

TOPIC		
ID	TEXT	PARENT_ID

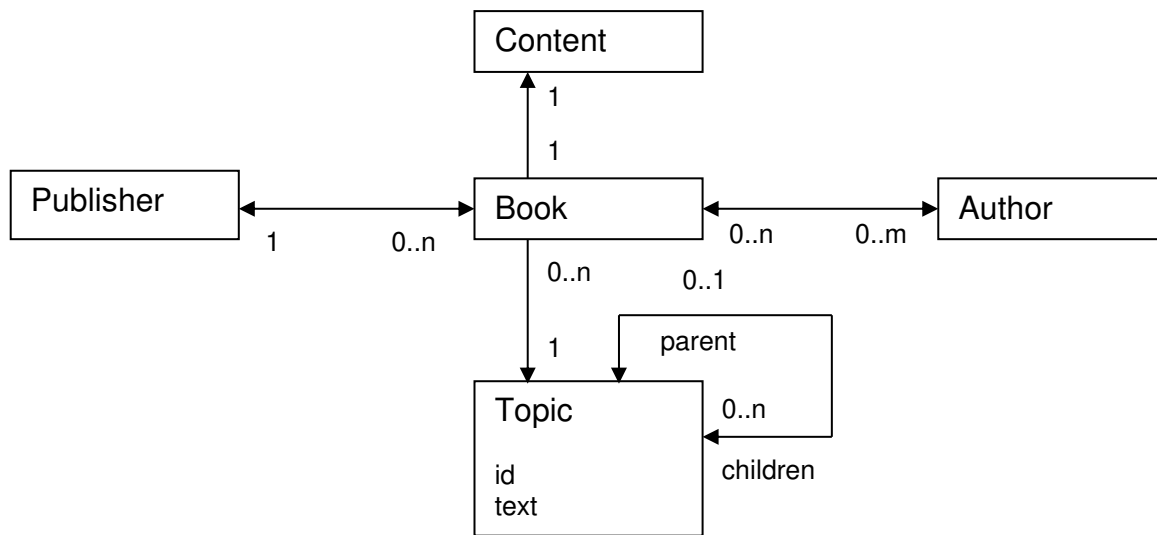
1	Alle Themen	NULL
2	Informatik	1
3	Comilerbau	2
4	Betriebssysteme	2
5	Programmiersprachen	2
6	Pascal	5
7	Modula	5
8	Oberon	5
9	Fremdsprachen	1
10	Latein	9
11	Altgriechisch	9

7.14 Alltogether

Im letzten Abschnitt dieses Kapitels soll ein "größeres" System vorgestellt werden: es geht um Authors, Books, Publishers, Contents und Topics.

Es geht u.a. um komplexe `join fetch` Abfragen.

Klassendiagramm



Man beachte, dass zwei Beziehungen unidirektional sind. Alle anderen Beziehungen sind bidirektional.

create.sql

```

create table CONTENT (
  ID integer generated by default as identity (start with 1),
  TEXT varchar (1024) not null,
  primary key (ID)
);

create table TOPIC (
  ID integer generated by default as identity (start with 1),
  TEXT varchar (20) not null,
  PARENT_ID integer,
  primary key (ID),
  foreign key (PARENT_ID) references TOPIC
);

create table PUBLISHER (
  ID integer generated by default as identity (start with 1),

```

```

    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table AUTHOR (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    CONTENT_ID integer,
    TOPIC_ID integer,
    PUBLISHER_ID integer,
    primary key (ID),
    unique (ISBN),
    unique (CONTENT_ID),
    foreign key (CONTENT_ID) references CONTENT,
    foreign key (TOPIC_ID) references TOPIC,
    foreign key (PUBLISHER_ID) references PUBLISHER
);

create table BOOK_AUTHOR (
    BOOKS_ID integer,
    AUTHORS_ID integer,
    primary key (BOOKS_ID, AUTHORS_ID),
    foreign key (BOOKS_ID) references BOOK,
    foreign key (AUTHORS_ID) references AUTHOR
);

insert into CONTENT (text) values('Programmiersprache Pascal');
insert into CONTENT (text) values('Programmiersprache Modula');
insert into CONTENT (text) values('Programmiersprache Oberon');

insert into TOPIC (text, parent_id) values('Alle Themen', null);
insert into TOPIC (text, parent_id) values('Informatik', 1);
insert into TOPIC (text, parent_id) values('Comilerbau', 2);
insert into TOPIC (text, parent_id) values('Betriebssysteme', 2);
insert into TOPIC (text, parent_id) values('Programmiersprachen', 2);
insert into TOPIC (text, parent_id) values('Pascal', 5);
insert into TOPIC (text, parent_id) values('Modula', 5);
insert into TOPIC (text, parent_id) values('Oberon', 5);
insert into TOPIC (text, parent_id) values('Fremdsprachen', 1);
insert into TOPIC (text, parent_id) values('Latein', 9);
insert into TOPIC (text, parent_id) values('Altgriechisch', 9);

insert into PUBLISHER (name) values('Addison');
insert into PUBLISHER (name) values('Prentice');

insert into AUTHOR (name) values('Wirth');

```

```
insert into AUTHOR (name) values('Meyer');

insert into BOOK (isbn, title, price, content_id, topic_id, publisher_id)
values('1111', 'Pascal', 10.0, 1, 6, 1);
insert into BOOK (isbn, title, price, content_id, topic_id, publisher_id)
values('2222', 'Modula', 20.0, 2, 7, 1);
insert into BOOK (isbn, title, price, content_id, topic_id, publisher_id)
values('3333', 'Oberon', 30.0, 3, 8, 2);

insert into BOOK_AUTHOR values(1, 1);
insert into BOOK_AUTHOR values(2, 1);
insert into BOOK_AUTHOR values(2, 2);
insert into BOOK_AUTHOR values(3, 1);
insert into BOOK_AUTHOR values(3, 2);
```

Man beachte, dass die Ausführung von `create.sql` die Tabellen nicht nur definiert, sondern sie zugleich auch füllt (und somit kann die `demoPersist`-Methode in der Anwendung entfallen).

Content

```
package domain;
// ...
@Entity
public class Content {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String text;

    // ...
}
```

Topic

```
package domain;
// ...
@Entity
public class Topic {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String text;

    @ManyToOne
    private Topic parent;
```

```
@OneToMany(mappedBy = "parent")
private Set<Topic> children = new HashSet<Topic>();

// ...

public void add(Topic topic) {
    this.children.add(topic);
    topic.parent = this;
}

// ...
}
```

Publisher

```
package domain;
// ...
@Entity
public class Publisher {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;

    @OneToMany(mappedBy = "publisher")
    private Set<Book> books = new HashSet<Book>();

    // ...
}
```

Author

```
package domain;
// ...
@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;

    @ManyToMany(mappedBy = "authors")
    private Set<Book> books = new HashSet<Book>();

    // ...
}
```

Book

```
package domain;
// ..
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    @OneToOne(fetch=FetchType.LAZY)
    private Content content;

    @ManyToOne(fetch=FetchType.LAZY)
    private Topic topic;

    @ManyToOne(fetch=FetchType.LAZY)
    private Publisher publisher;

    @ManyToMany(fetch=FetchType.LAZY)
    private Set<Author> authors = new HashSet<Author>();

    Book() { }

    public Book(String isbn, String title, double price,
                Content content, Topic topic, Publisher publisher,
                Author...authors) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
        this.topic = topic;
        this.content = content;
        this.publisher = publisher;
        this.publisher.getBooks().add(this);
        for (Author author : authors) {
            this.authors.add(author);
            author.getBooks().add(this);
        }
    }

    // ...
}
```


Application

```
static void demoAuthors(TransactionTemplate tt) {
    final String jpql =
        "select distinct a from Author a "
        + "join fetch a.books b "
        + "join fetch b.publisher "
        + "join fetch b.content "
        + "join fetch b.topic ";
    final List<Author> authors = tt.runWithResult(manager -> {
        final TypedQuery<Author> query =
            manager.createQuery(jpql, Author.class);
        return query.getResultList();
    });
    authors.forEach(author -> {
        System.out.println("AUTHOR = " + author);
        System.out.println("\tBOOKS [");
        author.getBooks().forEach(book -> {
            System.out.println("\t\tBOOK = " +
                book);
            System.out.println("\t\t\tCONTENT = " +
                book.getContent());
            System.out.println("\t\t\tTOPIC = " +
                book.getTopic());
            System.out.println("\t\t\tPUBLISHER" +
                book.getPublisher());
        });
        System.out.println("\t]");
    });
}
```

Die Ausgaben:

```
Hibernate: select ...
Hibernate: select ...
Hibernate: select ...
AUTHOR = Author [1, Wirth]
  BOOKS [
    BOOK = Book [3, 3333, 30.0, Oberon]
      CONTENT = Content [3, Programmiersprache Oberon]
      TOPIC = Topic [8, Oberon]
      PUBLISHERPublisher [2, Prentice]
    BOOK = Book [2, 2222, 20.0, Modula]
      CONTENT = Content [2, Programmiersprache Modula]
      TOPIC = Topic [7, Modula]
      PUBLISHERPublisher [1, Addison]
    BOOK = Book [1, 1111, 10.0, Pascal]
      CONTENT = Content [1, Programmiersprache Pascal]
      TOPIC = Topic [6, Pascal]
      PUBLISHERPublisher [1, Addison]
  ]
AUTHOR = Author [2, Meyer]
```

```

BOOKS [
    BOOK = Book [3, 3333, 30.0, Oberon]
    CONTENT = Content [3, Programmiersprache Oberon]
    TOPIC = Topic [8, Oberon]
    PUBLISHERPublisher [2, Prentice]
    BOOK = Book [2, 2222, 20.0, Modula]
    CONTENT = Content [2, Programmiersprache Modula]
    TOPIC = Topic [7, Modula]
    PUBLISHERPublisher [1, Addison]
]

```

```

static void demoBooks(TransactionTemplate tt) {
    final String jpql =
        "select distinct b from Book b "
        + "join fetch b.authors a "
        + "join fetch b.publisher "
        + "join fetch b.content "
        + "join fetch b.topic ";
    final List<Book> books = tt.runWithResult(manager -> {
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        return query.getResultList();
    });
    books.forEach(book -> {
        System.out.println("BOOK = " +
            book);
        System.out.println("\tCONTENT = " +
            book.getContent());
        System.out.println("\tTOPIC = " +
            book.getTopic());
        System.out.println("\tPUBLISHER" +
            book.getPublisher());
        System.out.println("\tAUTHORS [");
        book.getAuthors().forEach(author -> {
            System.out.println("\t\tAUTHOR = " +
                author);
        });
        System.out.println("\t]");
    });
}

```

Die Ausgaben:

```

Hibernate: select ...
Hibernate: select ...
Hibernate: select ...
BOOK = Book [1, 1111, 10.0, Pascal]
CONTENT = Content [1, Programmiersprache Pascal]
TOPIC = Topic [6, Pascal]
PUBLISHERPublisher [1, Addison]
AUTHORS [
    AUTHOR = Author [1, Wirth]
]

```

```

BOOK = Book [2, 2222, 20.0, Modula]
    CONTENT = Content [2, Programmiersprache Modula]
    TOPIC = Topic [7, Modula]
    PUBLISHER Publisher [1, Addison]
    AUTHORS [
        AUTHOR = Author [1, Wirth]
        AUTHOR = Author [2, Meyer]
    ]
BOOK = Book [3, 3333, 30.0, Oberon]
    CONTENT = Content [3, Programmiersprache Oberon]
    TOPIC = Topic [8, Oberon]
    PUBLISHER Publisher [2, Prentice]
    AUTHORS [
        AUTHOR = Author [1, Wirth]
        AUTHOR = Author [2, Meyer]
    ]

```

```

static void demoPublishers(TransactionTemplate tt) {
    final String jpql =
        "select distinct p from Publisher p "
        + "join fetch p.books b "
        + "join fetch b.content "
        + "join fetch b.topic "
        + "join fetch b.authors ";
    final List<Publisher> publishers = tt.runWithResult(
        manager -> {
            final TypedQuery<Publisher> query =
                manager.createQuery(jpql, Publisher.class);
            return query.getResultList();
        });
    publishers.forEach(publisher -> {
        System.out.println("PUBLISHER = " +
            publisher);
        System.out.println("\tBOOKS [");
        publisher.getBooks().forEach(book -> {
            System.out.println("\t\tBOOK = " +
                book);
            System.out.println("\t\t\tCONTENT = " +
                book.getContent());
            System.out.println("\t\t\tTOPIC = " +
                book.getTopic());
            System.out.println("\t\t\tAUTHORS [");
            book.getAuthors().forEach(author -> {
                System.out.println("\t\t\t\tAUTHOR = " +
                    author);
            });
            System.out.println("\t\t\t]");
        });
        System.out.println("\t]");
    });
}

```

Die Ausgaben:

```
Hibernate: select ...
Hibernate: select ...
Hibernate: select ...
PUBLISHER = Publisher [1, Addison]
  BOOKS [
    BOOK = Book [1, 1111, 10.0, Pascal]
    CONTENT = Content [1, Programmiersprache Pascal]
    TOPIC = Topic [6, Pascal]
    AUTHORS [
      AUTHOR = Author [1, Wirth]
    ]
    BOOK = Book [2, 2222, 20.0, Modula]
    CONTENT = Content [2, Programmiersprache Modula]
    TOPIC = Topic [7, Modula]
    AUTHORS [
      AUTHOR = Author [2, Meyer]
      AUTHOR = Author [1, Wirth]
    ]
  ]
PUBLISHER = Publisher [2, Prentice]
  BOOKS [
    BOOK = Book [3, 3333, 30.0, Oberon]
    CONTENT = Content [3, Programmiersprache Oberon]
    TOPIC = Topic [8, Oberon]
    AUTHORS [
      AUTHOR = Author [2, Meyer]
      AUTHOR = Author [1, Wirth]
    ]
  ]
]
```

7.15 Projection von Entities

Das Resultat eines `select`s mit Projektion ist - wie im Queries-Kapitel gezeigt - eine Liste von Arrays (oder ein einzelner Array). Bislang enthielt dieser Array nur relativ "dumme" Objekte - ein Objekt pro gelesenem Spaltenwert.

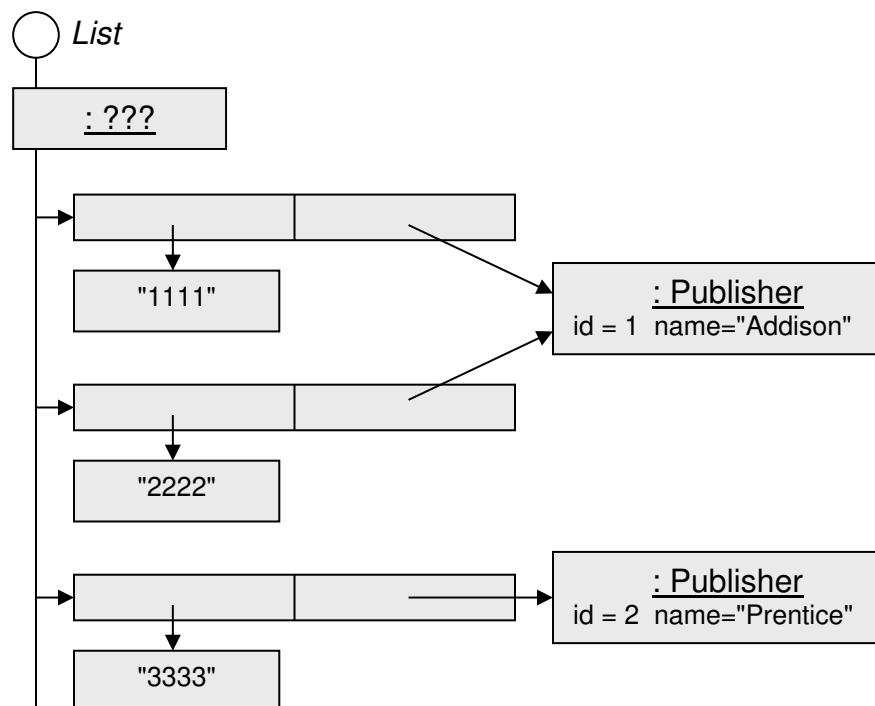
Ein solcher Array kann aber auch seinerseits wieder Objekte persistenter Klassen enthalten - wenn nämlich in der Projektionsliste ein nicht-primitives Attribut angesprochen wird: ein Attribut vom Typ einer persistenten Klasse.

In der folgenden Anwendung wird wieder das `Book-Publisher`-Beispiel benutzt - ein `Publisher` kann viele `Books` verlegen.

Man betrachte dann folgenden `select`:

```
select b.isbn, b.publisher from Book b
```

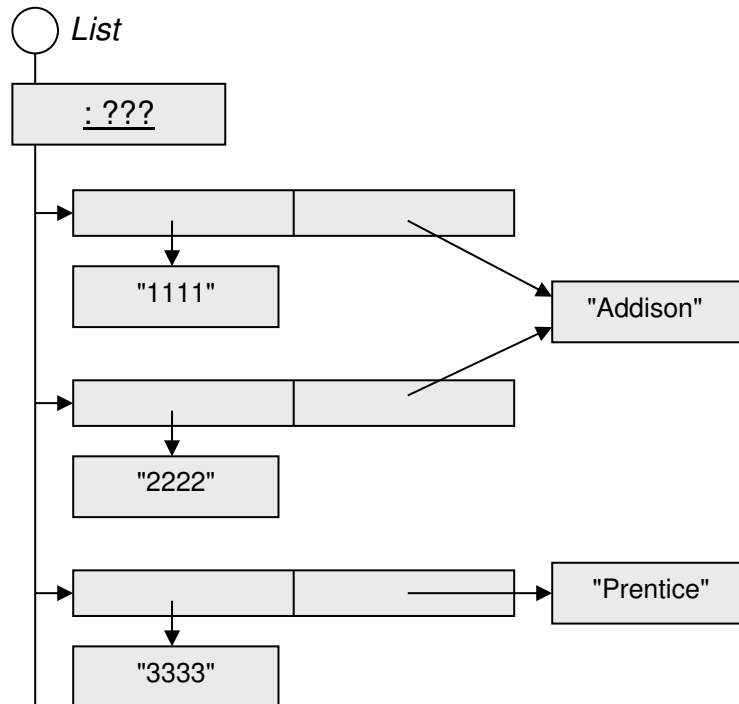
Das erste in der Projektion aufgezählte Attribut ist primitiv (ein Attribut vom Typ `String`); das zweite Attribut ist vom Typ `Publisher`. Das Resultat eines solchen `select`s könnte wie folgt aussehen:



Vielleicht möchte man aber neben der ISBN des Buches nur den Namen des Verlags - und nicht den "kompletten" Verlag. Dann könnte man folgenden `select` formulieren:

```
select b.isbn, b.publisher.name from Book b
```

Hier ein mögliches Resultat:



Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 10, p1));
        manager.persist(new Book("2222", "Modula", 20, p1));
        manager.persist(new Book("3333", "Oberon", 30, p2));
    });
}
```

```
static void demoQueryIsbnAndPublisher(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b.isbn, b.publisher from Book b";
        final TypedQuery<Object[]> query =
            manager.createQuery(jpql, Object[].class);
        final List<Object[]> rows = query.getResultList();
        for (final Object[] row : rows) {
            for (final Object value : row)
                System.out.println(value);
        }
    });
}
```

```

        System.out.println();
    };
});
}

```

Die Ausgaben:

```

Hibernate: select ... from Book b
         inner join Publisher p on b.publisher_id=p.id

```

1111

Publisher [1, Addison]

2222

Publisher [1, Addison]

3333

Publisher [2, Prentice]

```

static void demoQueryIsbnAndPublisherName(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b.isbn, b.publisher.name from Book b";
        final TypedQuery<Object[]> query =
            manager.createQuery(jpql, Object[].class);
        final List<Object[]> rows = query.getResultList();
        for (final Object[] row : rows) {
            for (final Object value : row)
                System.out.println(value);
            System.out.println();
        }
    });
}

```

Die Ausgaben:

```

Hibernate: select ... from Book b, Publisher p
         where b.publisher_id=p.id

```

1111

Addison

2222

Addison

3333

Prentice

7.16 Constructor Expressions mit Entities

Auch hier werden wieder die Klassen `Book` und `Publisher` verwendet - dieselben, die auch im letzten Abschnitt verwendet wurden.

Weiterhin gibt es eine ganz "dumme" Klasse, eine Klasse, die nicht(!) als `@Entity` gekennzeichnet ist - und somit keine(!) persistente Klasse ist. Objekte dieser Klasse sind einfach nur "komplexe Werte". Man beachte, dass sie weder getter / setter noch einen parameterlosen Konstruktor besitzt.

Sie besitzt stattdessen drei unterschiedlich parametrisierte Konstruktoren:

BookData

```
package appl;

public class BookData {
    // ...
    public String isbn;
    public String title;
    public String publisherName;

    public BookData(String isbn, String title, String publisherName) {
        this.isbn = isbn;
        this.title = title;
        this.publisherName = publisherName;
    }
    public BookData(String isbn, String title) {
        this(isbn, title, null);
    }
    public BookData(String title) {
        this(null, title, null);
    }

    @Override
    public String toString() { ... }
}
```

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 10, p1));
        manager.persist(new Book("2222", "Modula", 20, p1));
        manager.persist(new Book("3333", "Oberon", 30, p2));
    });
}
```



```
}
```

```
static void demoQueryIsbnTitlePublisherName(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select new appl.BookData(" +
            "b.isbn, b.title, b.publisher.name) from Book b";
        final TypedQuery<BookData> query =
            manager.createQuery(jpql, BookData.class);
        query.getResultList().forEach(System.out::println);
    });
}
```

Ausgaben:

```
Hibernate: select ... from Book b, Publisher p
           where b.publisher_id=p.id
appl.BookData [1111, Pascal, Addison]
appl.BookData [2222, Modula, Addison]
appl.BookData [3333, Oberon, Prentice]
```

```
static void demoQueryIsbnTitle(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select new appl.BookData(" +
            "b.isbn, b.title) from Book b";
        final TypedQuery<BookData> query =
            manager.createQuery(jpql, BookData.class);
        query.getResultList().forEach(System.out::println);
    });
}
```

Ausgaben:

```
Hibernate: select ... from Book b
appl.BookData [1111, Pascal, null]
appl.BookData [2222, Modula, null]
appl.BookData [3333, Oberon, null]
```

```
static void demoQueryTitle(TransactionTemplate tt) {
    Util.mlog();
    tt.run(manager -> {
        final String jpql =
            "select new appl.BookData(" +
            "b.title) from Book b";
        final TypedQuery<BookData> query =
            manager.createQuery(jpql, BookData.class);
        query.getResultList().forEach(System.out::println);
    });
}
```

Ausgaben:

```
Hibernate: select ... from Book b
appl.BookData [null, Pascal, null]
appl.BookData [null, Modula, null]
appl.BookData [null, Oberon, null]
```

7.17 Views vs. Join-Fetch – Performance

JPA kann auch auf Views zugreifen (zumindest lesend). Eine View wird von JPA einfach als gewöhnliche Tabelle gesehen. Zeilen einer View können also auf `@Entity`-Objekte abgebildet werden.

create.sql

```
create table CONTENT (  
    ID integer generated by default as identity (start with 1),  
    TEXT varchar (1024) not null,  
    primary key (ID)  
);  
  
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    ISBN varchar (20) not null,  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    CONTENT_ID integer,  
    primary key (ID),  
    unique (ISBN),  
    unique (CONTENT_ID),  
    foreign key (CONTENT_ID) references CONTENT  
);  
  
create view BOOKCONTENT as  
    select  
        id, isbn, price, title, text  
    from  
        Book b  
    inner join  
        Content c on b.content_id=c.id
```

views.BookContent

```
package views;  
// ...  
@Entity  
public class BookContent {  
  
    @Id  
    private Integer id;  
  
    @Basic  
    private String isbn;  
  
    @Basic  
    private String title;
```

```

@Basic
private double price;

@Basic
private String text;

// getter, setter, toString...
}

```

appl.Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        for (int i = 0; i < 500; i++) {
            final Content c = new Content("Pascal-Language" + i);
            final Book b = new Book("1111" + i, "Pascal", 10, c);
            manager.persist(c);
            manager.persist(b);
        }
    });
}

```

```

static void demoPerformance(TransactionTemplate tt) {
    tt.runPerfomanceTest(500,
        manager -> {
            final String jpql =
                "select b from Book b inner join fetch b.content";
            final TypedQuery<Book> query =
                manager.createQuery(jpql, Book.class);
            query.getResultList().forEach(
                book -> book.getContent().getText());
        },
        manager -> {
            final String jpql =
                "select bc from BookContent bc";
            final TypedQuery<BookContent> query =
                manager.createQuery(jpql, BookContent.class);
            query.getResultList().forEach(
                bookContent -> bookContent.getText());
        }
    );
}

```

Die Ausgaben:

```

duration[0] = 2431 milliseconds
duration[1] = 1675 milliseconds

```

Der Zugriff auf Views ist performanter als der Join-Fetch.

7.18 Abbildung von HashMaps

Im folgenden wird eine sehr spezielle Art von Abbildungen vorgestellt. Die Anwendung sei dem Leser / der Leserin zum Selbststudium überlassen...

Die Anwendung handelt von Schulklassen (Forms), Fächern (Subjects) und Lehrern (Teachers).

create.sql

```
CREATE TABLE SUBJECT (
    ID integer generated by default as identity (start with 1),
    NAME VARCHAR (20) NOT NULL,
    PRIMARY KEY (ID)
);

CREATE TABLE TEACHER (
    ID integer generated by default as identity (start with 1),
    NAME VARCHAR (20) NOT NULL,
    PRIMARY KEY (ID),
);

CREATE TABLE FORM (
    ID integer generated by default as identity (start with 1),
    NAME VARCHAR (20) NOT NULL,
    PRIMARY KEY (ID)
);

CREATE TABLE FORM_SUBJECT_TEACHER (
    FORM_ID integer NOT NULL,
    SUBJECT_ID integer NOT NULL,
    TEACHER_ID integer NOT NULL,
    PRIMARY KEY (FORM_ID, SUBJECT_ID, TEACHER_ID),
    foreign key (FORM_ID) references FORM,
    foreign key (SUBJECT_ID) references SUBJECT,
    foreign key (TEACHER_ID) references TEACHER
);
```

Subject

```
package domain;
// ...
@Entity
public class Subject {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;
```

```
// Konstruktoren, gettter, setter, toString ...

@Override
public int hashCode() { ... }

@Override
public boolean equals(Object obj) { ... }
}
```

Subjects werden in der Anwendung als Schlüssel in `HashMaps` benutzt werden. Daher die Notwendigkeit, `hashCode` und `equals` zu überschreiben...

Teacher

```
package domain;
// ...
@Entity
public class Teacher {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;

    // Konstruktoren, gettter, setter, toString ...
}
```

Form

```
package domain;
// ...
@Entity
public class Form {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;

    @OneToMany
    @JoinTable(
        name = "FORM_SUBJECT_TEACHER",
        joinColumns = @JoinColumn(name = "TEACHER_ID"),
        inverseJoinColumns = @JoinColumn(name = "FORM_ID"))
    @MapKeyJoinColumn(name = "SUBJECT_ID")
    private Map<Subject, Teacher> subjectTeacherMap = new HashMap<>();

    // Konstruktoren, gettter, setter, toString ...
}
```

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {

        final Subject mathe = new Subject("mathe");
        final Subject deutsch = new Subject("deutsch");
        final Subject englisch = new Subject("englisch");

        Teacher broer = new Teacher("broer");
        Teacher lueke = new Teacher("lueke");

        final Form klasse6= new Form("klasse6");
        final Form klasse8= new Form("klasse8");

        manager.persist(mathe);
        manager.persist(deutsch);
        manager.persist(englisch);

        manager.persist(broer);
        manager.persist(lueke);

        manager.persist(klasse6);
        manager.persist(klasse8);

        klasse6.getSubjectTeacherMap().put(mathe, broer);
        klasse6.getSubjectTeacherMap().put(englisch, broer);
        klasse6.getSubjectTeacherMap().put(deutsch, lueke);

        klasse8.getSubjectTeacherMap().put(mathe, lueke);
        klasse8.getSubjectTeacherMap().put(englisch, broer);
    });
}

```

Nach Ausführung der obigen Transaktion sieht die Datenbank wie folgt aus:

```

SUBJECT
ID  NAME
-----
1   mathe
2   deutsch
3   englisch
-----

```

```

TEACHER
ID  NAME
-----
1   broer
2   lueke
-----

```

```

FORM

```

ID NAME

```
-----
1  klasse6
2  klasse8
-----
```

FORM_SUBJECT_TEACHER

FORM_ID SUBJECT_ID TEACHER_ID

```
-----
1      1      1
1      3      1
1      3      2
2      1      2
2      2      1
-----
```

Eine Query-Transaktion:

```
static void demoQuery(TransactionalTemplate tt) {
    tt.run(manager -> {
        // String jpql = "select distinct f from Form f " +
        //               "join fetch f.subjectTeacherMap";
        String jpql = "select f from Form f";
        TypedQuery<Form> query = manager.createQuery(jpql, Form.class);
        List<Form> forms = query.getResultList();
        forms.forEach(f -> {
            System.out.println(f);
            f.getSubjectTeacherMap().forEach((k, v) ->
                System.out.println("\t" + k + " ==> " + v));
        });
    });
}
```

Die Ausgaben:

Form [id=1, name=klasse6]

```
Subject [id=1, name=mathe] ==> Teacher [id=1, name=broer]
Subject [id=3, name=englisch] ==> Teacher [id=1, name=broer]
Subject [id=2, name=deutsch] ==> Teacher [id=2, name=lueke]
```

Form [id=2, name=klasse8]

```
Subject [id=1, name=mathe] ==> Teacher [id=2, name=lueke]
Subject [id=3, name=englisch] ==> Teacher [id=1, name=broer]
```

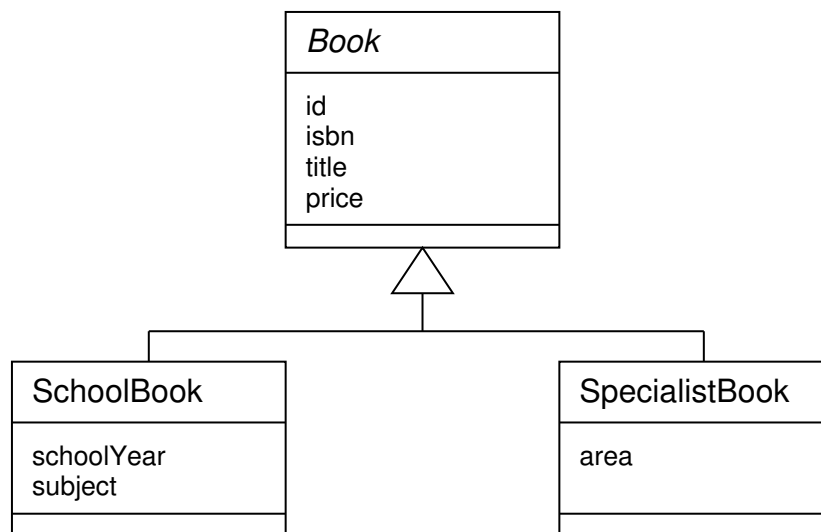

8 Vererbung

Vererbungsbeziehungen können mit JPA auf drei unterschiedliche Arten auf Tabellen abgebildet werden:

- Eine einzige Tabelle kann die Daten aller Objekte aller Klassen einer Vererbungshierarchie enthalten (`SINGLE_TABLE`).
- Pro Klasse der Vererbungshierarchie existiert genau eine Tabelle. Die Tabellen für die abgeleiteten Klassen "erben" dann von der Basisklassentabelle (`JOINED`).
- Nur für jede instantiierbare Klasse existiert eine Tabelle. Für die (abstrakte) Basisklasse existiert keine eigene Tabelle (`TABLE_PER_CLASS`).

Die Abbildung von Vererbungsbeziehungen auf Tabellen muss Polymorphie unterstützen. Es muss also möglich sein, eine Liste aller Objekte aller abgeleiteten Klassen abzufragen. Diese Liste muss dann natürlich polymorph sein (muss eben Objekte unterschiedlichen, aber bezüglich des Basistyps verwandten Typs beinhalten).

Für die folgenden Abschnitte wird folgende Vererbungsbeziehung zugrunde gelegt: Ein Buch ist entweder ein Fachbuch (`SpecialistBook`) oder ein Schulbuch (`SchoolBook`). Die Basisklasse `Book` ist abstrakt - nur `SpecialistBook` und `SchoolBook` sind instanzierbar. In der `Book`-Klasse sind bereits die folgenden Attribute und Properties vereinbart: die (generierte) `id`, die `isbn`-Nummer (als business key), der `title` und der `price`. Die Klasse `SchoolBook` definiert zwei weitere Attribute und Properties: die Jahrgangsstufe (`year`) und das Thema (`subject`). Die Klasse `SpecialistBook` definiert zusätzlich das Fachgebiet (`area`).



In den ersten drei Abschnitten dieses Kapitels werden dieselben Java-Klassen verwendet. Sie unterscheiden sich nur in den Annotations, welche das jeweils spezifische Mapping beschreiben.

Book

```
package domain;
// ...
@Entity
// specific inheritance mapping
public abstract class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String isbn;

    @Basic
    private String title;

    @Basic
    private double price;

    // Konstruktoren, getter, setter, toString...
}
```

Bei einer Basisklasse ist es entscheidend, dass in ihr bereits die `id` und auch der business key (hier: `isbn`) existieren (und somit auch die `equals`- und `hashCode`-Methoden bereits endgültig implementierbar sind).

SchoolBook

```
package domain;
// ...
@Entity
// specific inheritance mapping
public class SchoolBook extends Book {

    @Basic
    private int schoolYear;

    @Basic
    private String subject;

    // Konstruktoren, getter, setter, toString...
}
```

SpecialistBook

```
package domain;
```

```
// ...
@Entity
// specific inheritance mapping
public class SpecialistBook extends Book {

    @Basic
    private String area;

    // Konstruktoren, getter, setter, toString...
}
```

Auch die Application ist bei allen drei Varianten dieselbe:

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new SchoolBook(
            "1111", "Mathe5", 2000, 5, "Mathematics"));
        manager.persist(new SchoolBook(
            "2222", "RedLine", 2500, 5, "English"));
        manager.persist(new SpecialistBook(
            "3333", "Java", 5000, "Computer"));
    });
}
```

```
static void demoQuerySchoolBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select b from SchoolBook b";
        final TypedQuery<SchoolBook> query =
            manager.createQuery(jpql, SchoolBook.class);
        query.getResultList().forEach(System.out::println);
    });
}
```

Die Ausgaben:

```
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
```

```
static void demoQuerySpecialistBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select b from SpecialistBook b";
        final TypedQuery<SpecialistBook> query =
            manager.createQuery(jpql, SpecialistBook.class);
        query.getResultList().forEach(System.out::println);
    });
}
```

Die Ausgaben:

```
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
```

```
static void demoQueryBooks(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql = "select b from Book b";  
        final TypedQuery<Book> query =  
            manager.createQuery(jpql, Book.class);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

Die Ausgaben:

```
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]  
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]  
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
```

Man beachte, dass `demoQueryBooks` **alle(!)** Books liefert.

8.1 Single Table

Bei der Abbildung auf eine einzige Tabelle muss für alle Attribute / Properties der Basisklasse und für alle Attribute / Properties aller abgeleiteten Klassen jeweils eine Tabellenspalte existieren. D.h., dass bei der Abbildung eines Objekts auf eine Zeile i.d.R. einige Spalten leer bleiben. Deshalb müssen alle Spalten, die für die Spezial-Attribute / -Properties der abgeleiteten Klassen eingerichtet werden, `NULL`-Wert fähig sein.

Zusätzlich muss eine solche Tabelle eine Spalte enthalten, aus deren Wert hervorgeht, um welchen Typ es sich bei der aktuellen Zeile handelt (hier also: `SchoolBook` oder `SpecialistBook`). Eine solche Spalte wird als Diskriminator-Spalte bezeichnet. Der Default-Name dieser Spalte ist `DTYPE`. Für jede instanziiierbare Klasse muss es für diese Spalte einen eindeutigen Wert geben. Wiederum per Default ist dies der nicht qualifizierte Name der Klasse. Natürlich kann man diese Defaults auch umgehen.

create.sql

```
create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    DTYPE varchar (32),
    SCHOOLYEAR integer,
    SUBJECT varchar (128),
    AREA varchar (128),
    primary key (ID),
    unique (ISBN)
)
```

Und hier die spezifischen Mappings:

Book

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
//@DiscriminatorColumn(name="DTYPE", // this is the default
//                      discriminatorType=DiscriminatorType.STRING)

public class Book ...
```

SchoolBook

```
@Entity
//@DiscriminatorValue("SchoolBook") // this is the default
public class SchoolBook extends Book ...
```

SpecialistBook

```
@Entity
//@DiscriminatorValue("SpecialistBook") // this is the default
public class SpecialistBook extends Book ...
```

Das Resultat:

```
BOOK
ID ISBN  TITLE  PRICE  DTYPE
      SCHOOLYEAR SUBJECT  AREA
-----
1  1111 Mathe5   2000.0 SchoolBook
      5         Mathematics NULL
2  2222 RedLine 2500.0 SchoolBook
      5         English    NULL
3  3333 Java    5000.0 SpecialistBook
      NULL      NULL      Computer'
-----
```

Die Ausgabe:

```
+-----+
| demoPersist
+-----+
Hibernate: insert into Book (...) values (...)
Hibernate: insert into Book (...) values (...)
Hibernate: insert into Book (...) values (...)
+-----+
| demoQuerySchoolBooks
+-----+
Hibernate: select b.id, b.isbn, b.price, b.title,
      b.schoolYear, b.subject
      from Book b where b.DTYPE='SchoolBook'
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
+-----+
| demoQuerySpecialistBooks
+-----+
Hibernate: select b.id, b.isbn, b.price, b.title,
      b.area
      from Book b where b.DTYPE='SpecialistBook'
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
+-----+
| demoQueryBooks
+-----+
Hibernate: select b.id, b.isbn, b.price, b.title,
      b.area, b.schoolYear, b.subject,
      b.DTYPE
      from Book b
```

```
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]  
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]  
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
```

8.2 Joined-subclass

Beim `joined-subclass`-Mapping wird eine Tabelle für die Basisklasse eingerichtet und für jede abgeleitete Klasse eine weitere Tabelle.

create.sql

```
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    ISBN varchar (20) not null,  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    primary key (ID),  
    unique (ISBN)  
);  
  
create table SCHOOLBOOK (  
    ID integer  
    SCHOOLYEAR integer,  
    SUBJECT varchar (128),  
    primary key (ID),  
    foreign key (ID) references BOOK  
);  
  
create table SPECIALISTBOOK (  
    ID integer  
    AREA varchar (128),  
    primary key (ID),  
    foreign key (ID) references BOOK  
);
```

Man beachte, dass `YEAR` / `SUBJECT` resp. `AREA` hier wieder als `not null` spezifiziert werden könnten.

Man beachte weiterhin, dass die `ID`-Spalte der "abgeleiteten" Tabellen sowohl deren Primary Key als auch der Fremdschlüssel zur `BOOK`-Tabelle ist.

Und man beachte schließlich, dass es keine Diskriminator-Spalte mehr gibt.

Hier die spezifischen Mappings:

Book

```
@Entity  
@Inheritance(strategy=InheritanceType.JOINED)  
public class Book...
```


SchoolBook

```
@Entity
public class SchoolBook extends Book ...
```

SpecialistBook

```
@Entity
public class SpecialistBook extends Book ...
```

Die abgeleiteten Klassen benötigen kein weiteres Mapping.

Der Zustand der Datenbank:

```
BOOK
ID ISBN TITLE PRICE
-----
1 1111 Mathe5 2000.0
2 2222 RedLine 2500.0
3 3333 Java 5000.0
-----

SCHOOLBOOK
ID SCHOOLYEAR SUBJECT
-----
1 5 Mathematics
2 5 English
-----

SPECIALISTBOOK
ID AREA
-----
3 Computer
-----
```

Die Ausgabe des Programms:

```
+-----+
| demoPersist
+-----+
Hibernate: insert into Book (...) values (default, ?, ?, ?)
Hibernate: insert into SchoolBook (...) values (?, ?, ?)
Hibernate: insert into Book (...) values (default, ?, ?, ?)
Hibernate: insert into SchoolBook (...) values (?, ?, ?)
Hibernate: insert into Book (...) values (default, ?, ?, ?)
Hibernate: insert into SpecialistBook (...) values (?, ?)
+-----+
| demoQuerySchoolBooks
+-----+
Hibernate: select sb.id, b.isbn, b.price, b.title,
```

```

        sb.schoolYear, sb.subject
    from SchoolBook sb
    inner join Book b on sb.id=s.id
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
+-----+
| demoQuerySpecialistBooks
+-----+
Hibernate: select sb.id, b.isbn, b.price, b.title,
        sb.area
        from SpecialistBook sb
        inner join Book b on sb.id=b.id
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
+-----+
| demoQueryBooks
+-----+
Hibernate: select b.id, b.isbn, b.price, b.title,
        b1.area,
        b2.schoolYear, b2.subject,
        case
            when b1.id is not null then 1
            when b2.id is not null then 2
            when b.id is not null then 0
        end as clazz
        from Book b
        left outer join SpecialistBook b1 on b.id=b1.id
        left outer join SchoolBook b2 on b.id=b2.id
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
SpecialistBook [Computer, 3, 3333, 5000.0, Java]

```

8.3 Class per Table

Beim `CLASS-PER-TABLE`-Mapping wird nur für jede instantiierbare Klasse eine Tabelle angelegt – für die abstrakte Basisklasse existiert also keine eigene Tabelle (ein Mapping für diese Klasse muss allerdings angelegt werden). Für das `Book`-Beispiel werden also nur zwei Tabellen existieren: eine Tabelle für die `SchoolBooks` und eine zweite für die `SpecialistBooks`.

Bei dieser Variante muss eine tabellenbasierte ID-Generierung vorgesehen werden (um für beide Tabellen einen einheitlichen ID-Kreis zu erzwingen).

create.sql

```
create table SCHOOLBOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    SCHOOLYEAR integer,
    SUBJECT varchar (128),
    primary key (ID),
    unique (ISBN)
);

create table SPECIALISTBOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    AREA varchar (128),
    primary key (ID),
    unique (ISBN)
);

create table SEQUENCE (
    SEQ_NAME varchar (255),
    SEQ_COUNT integer,
    primary key (SEQ_NAME)
);

insert into SEQUENCE values ('BOOK', 0);
```

Book

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Book {
    // ...
    @Id
    @TableGenerator(name = "tableGenerator", table = "SEQUENCE",
        pkColumnName = "SEQ_NAME", pkColumnValue = "BOOK",
```

```

        valueColumnName = "SEQ_COUNT", allocationSize = 20)
    @GeneratedValue(strategy = GenerationType.TABLE,
        generator = "tableGenerator")
    public Integer getId() { ... }
    void setId(Integer id) { ... }
    // ...
}

```

SchoolBook

```

@Entity
public class SchoolBook extends Book ...

```

SpecialistBook

```

@Entity
public class SpecialistBook extends Book ...

```

Die abgeleiteten Klassen benötigen kein weiteres Mapping.

Hier das Resultat:

```

SCHOOLBOOK
ID ISBN TITLE    PRICE  SCHOOLYEAR SUBJECT
-----
1  1111 Mathe5   2000.0 5      Mathematics
2  2222 RedLine 2500.0 5      English
-----

SPECIALISTBOOK
ID ISBN TITLE PRICE  AREA
-----
3  3333 Java   5000.0 Computer
-----

SEQUENCE
SEQ_NAME SEQ_COUNT
-----
BOOK      1
-----

```

Die Ausgaben:

```

+-----+
| demoPersist
+-----+
Hibernate: select SEQ_COUNT from SEQUENCE
           where SEQ_NAME = 'BOOK' for update
Hibernate: update SEQUENCE set SEQ_COUNT = ?
           where SEQ_COUNT = ? and SEQ_NAME = 'BOOK'

```

```

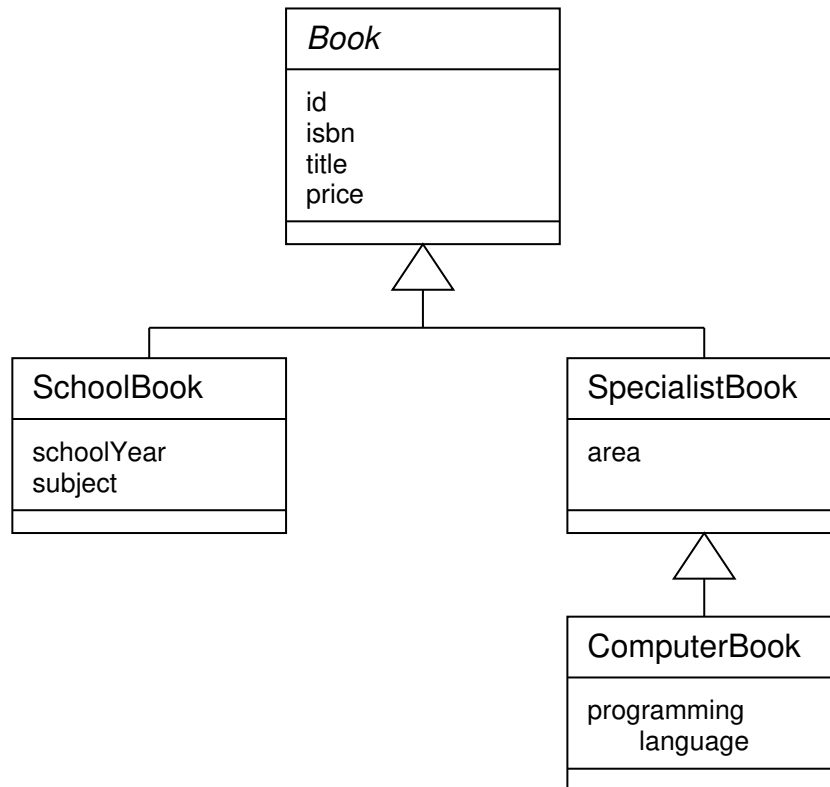
Hibernate: insert into SchoolBook (...) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into SchoolBook (...) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into SpecialistBook (...) values (?, ?, ?, ?, ?, ?)
+-----+
| demoQuerySchoolBooks
+-----+
Hibernate: select b.id, b.isbn, b.price, b.title,
               b.schoolYear, b.subject
               from SchoolBook b
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
+-----+
| demoQuerySpecialistBooks
+-----+
Hibernate: select b.id, b.isbn, b.price, b.title,
               b.area
               from SpecialistBook b
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
+-----+
| demoQueryBooks
+-----+
Hibernate: select b.id, b.isbn, b.price, b.title,
               b.area, b.schoolYear, b.subject, b.clazz
               from (
                   select id, isbn, price, title,
                          area,
                          cast(null as int) as schoolYear,
                          cast(null as varchar(100)) as subject,
                          1 as clazz_
                   from SpecialistBook
                   union all select id, isbn, price, title,
                          cast(null as varchar(100)) as area,
                          schoolYear, subject, 2 as clazz_ from SchoolBook
               ) b
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]

```

8.4 Tiefe Vererbung

Natürlich können wir mit JPA auch tiefere Vererbungshierarchien abbilden.

Die `Book`-Klassen werden um eine Klasse `ComputerBook` erweitert, die von `SpecialistBook` abgeleitet ist:



Als Abbildungsvariante verwenden wird `joined-subclass`.

Das Datenbank-Schema enthält also vier Tabellen:

create.sql

```
create table BOOK ( ... );
create table SCHOOLBOOK ( ... );
create table SPECIALISTBOOK ( ... );
create table COMPUTERBOOK (
    ID integer ,
    PROGRAMMINGLANGUAGE varchar(64) ,
```

```
primary key (ID),
foreign key (ID) references SPECIALISTBOOK
);
```

(Der Fremdschlüssel von `COMPUTERBOOK` kann sich sowohl auf `SPECIALISTBOOK` als auch auf `BOOK` beziehen. Die "Vererbung" wird aber deutlicher, wenn `SPECIALISTBOOK` als Referenz-Tabelle verwendet wird.)

Hier die zusätzliche Klasse `ComputerBook`:

ComputerBook

```
package domain;
// ...
@Entity
public class ComputerBook extends SpecialistBook {

    @Basic
    private String programmingLanguage;

    // Konstruktoren, getter, setter, toString...
}
```

Application

Die Persist-Transaktion persistiert zwei `SchoolBooks`, ein `SpecialistBook` und ein `ComputerBook`:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new SchoolBook(
            "1111", "Mathe5", 2000, 5, "Mathematics"));
        manager.persist(new SchoolBook(
            "2222", "RedLine", 2500, 5, "English"));
        manager.persist(new SpecialistBook(
            "3333", "Java", 5000, "Computer"));
        manager.persist(new ComputerBook(
            "4444", "Java", 5000, "COMPUTER", "Pascal"));
    });
}
```

Wie selektieren alle `SchookBooks`:

```
static void demoQuerySchoolBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select b from SchoolBook b";
        final TypedQuery<SchoolBook> query =
            manager.createQuery(jpql, SchoolBook.class);
        query.getResultList().forEach(System.out::println);
    });
}
```

SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]

Wir selektieren alle **SpecialistBooks** – und erhalten zwei(!) Bücher:

```
static void demoQuerySpecialistBooks(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql = "select b from SpecialistBook b";  
        final TypedQuery<SpecialistBook> query =  
            manager.createQuery(jpql, SpecialistBook.class);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

SpecialistBook [Computer, 3, 3333, 5000.0, Java]
ComputerBook [COMPUTER, 4, 4444, 5000.0, Pascal, Java]

Wie selektieren nur die **ComputerBooks**:

```
static void demoQueryComputerBooks(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql = "select b from ComputerBook b";  
        final TypedQuery<ComputerBook> query =  
            manager.createQuery(jpql, ComputerBook.class);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

ComputerBook [COMPUTER, 4, 4444, 5000.0, Pascal, Java]

Uns schließlich selektieren wir alle **Books**:

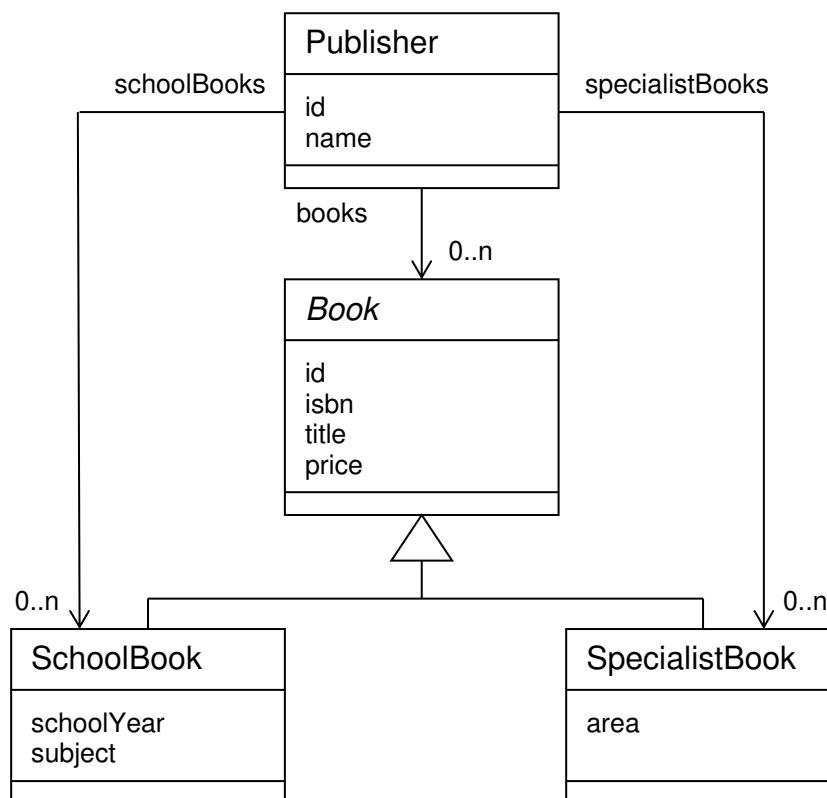
```
static void demoQueryBooks(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql = "select b from Book b";  
        final TypedQuery<Book> query =  
            manager.createQuery(jpql, Book.class);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
ComputerBook [COMPUTER, 4, 4444, 5000.0, Pascal, Java]

8.5 Vererbung und Assoziationen

Angenommen, ein `Publisher` hat drei Collections: `books`, `schoolBooks` und `specialistBooks`. Das `books`-Attribut sei vom Typ `List`, die Attribute `schoolBook` und `specialistBook` vom Typ `Set`.

Wir könnten uns natürlich auf eine einzige Liste beschränken: auf die Liste aller `Books` des jeweiligen `Publisher`. Sind wir dann aber z.B. ausschließlich an `SchoolBooks` interessiert, müssten wir die erhaltenen `Books` jeweils auf `SchoolBook` downcasten. Genau dieses Problem soll durch die Einführung zweier spezieller Collections vermieden werden.



Wir verwenden die Single-Table-Abbildung der `Book`-Klassenhierarchie:

Book, SchoolBook, SpecialistBook

```

package domain;
// ...
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
  
```

```
public class Book { ... }
```

```
package domain;
// ...
@Entity
public class SchoolBook extends Book { ... }
```

```
package domain;
// ...
@Entity
public class SpecialistBook extends Book { ... }
```

Publisher

```
package domain;
// ...
@Entity
public class Publisher {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String name;

    @OneToMany(fetch = FetchType.LAZY)
    @JoinColumn(name = "PUBLISHER_ID") // required!!!
    private List<Book> books =
        new ArrayList<Book>();

    @OneToMany(fetch = FetchType.LAZY)
    @JoinColumn(name = "PUBLISHER_ID") // required!!!
    private Set<SchoolBook> schoolBooks =
        new HashSet<SchoolBook>();

    @OneToMany(fetch = FetchType.LAZY)
    @JoinColumn(name = "PUBLISHER_ID") // required!!!
    private Set<SpecialistBook> specialistBooks =
        new HashSet<SpecialistBook>();

    // Konstruktoren, getter, setter, toString ...
}
```

(Man beachte, dass das `books`-Attribut vom Typ `List` ist, die `specialistBooks`- und `schoolBooks`-Attribute aber vom Typ `Set`. Wären auch diese beiden Attribute vom Typ `List`, würde das letzte der folgenden Query-Beispiele nicht funktionieren...)

Application

Wir persistieren einen `Publisher` und drei `Books`. Jedes erzeugte `Book` wird in die `books`-Liste des `Publishers` übernommen:

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        Publisher p = new Publisher("Addison");
        manager.persist(p);

        SchoolBook b1 = new SchoolBook(
            "1111", "Mathe5", 2000, 5, "Mathematics");
        p.getSchoolBooks().add(b1);
        manager.persist(b1);

        SchoolBook b2 = new SchoolBook(
            "2222", "RedLine", 2500, 5, "English");
        p.getSchoolBooks().add(b2);
        manager.persist(b2);

        SpecialistBook b3 = new SpecialistBook(
            "3333", "Java", 5000, "Computer");
        p.getSpecialistBooks().add(b3);
        manager.persist(b3);
    });
}

```

Es existieren nun drei **BOOK**-Zeilen, die alle dieselbe **PUBLISHER**-Zeile referenzieren.

Wir selektieren die **SchookBooks** des **Publishers** – und lesen diese **SchoolBooks** aus der **schoolBooks-Collection** aus::

```

static void demoQuerySchoolBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select p from Publisher p " +
            "join fetch p.schoolBooks b " +
            "where p.id = 1";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        final Publisher p = query.getSingleResult();

        p.getSchoolBooks().forEach(System.out::println);
    });
}

```

```

SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]

```

Wir selektieren die **SpecialistBooks** des **Publishers** – und lesen diese **SpecialistBooks** aus der **specialistBooks-Collection** aus::

```

static void demoQuerySpecialistBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select p from Publisher p " +
            "join fetch p.specialistBooks b " +
            "where p.id = 1";
        final TypedQuery<Publisher> query =

```

```

        manager.createQuery(jpql, Publisher.class);
        final Publisher p = query.getSingleResult();

        p.getSpecialistBooks().forEach(System.out::println);
    });
}

```

SpecialistBook [Computer, 3, 3333, 5000.0, Java]

Wir selektieren alle Books des Publishers – und lesen die Books aus der books-Collection aus. Und wir lesen zusätzlich die SchoolBooks aus der schoolBooks-Collection und die SpecialistBooks aus der specialistBooks-Collection aus:

```

static void demoQueryBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql = "select p from Publisher p " +
            "join fetch p.books b " +
            "where p.id = 1";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        final Publisher p = query.getSingleResult();

        p.getBooks().forEach(System.out::println);
        p.getSchoolBooks().forEach(System.out::println); // lazy
        p.getSpecialistBooks().forEach(System.out::println); // lazy
    });
}

```

SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
SpecialistBook [Computer, 3, 3333, 5000.0, Java]

SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]

SpecialistBook [Computer, 3, 3333, 5000.0, Java]

Die letzten beiden Aufrufe (getSchoolBooks und getSpecialistBooks) führen jeweils zu einem neuen Lazy-Loading!

Wie können wir diese zusätzlichen Lazy-Loadings vermeiden? Wir benutzen zwei join-fetch-Klauseln:

```

static void demoQuerySchoolBooksAndSpecialistBooks(
    TransactionTemplate tt) {
    tt.run(manager -> {
        // das geht nur bei Sets...
        final String jpql = "select p from Publisher p " +
            "join fetch p.schoolBooks " +
            "join fetch p.specialistBooks " +
            "where p.id = 1";
        final TypedQuery<Publisher> query = manager.createQuery(

```

```
        jpql, Publisher.class);
        final Publisher p = query.getSingleResult();

        p.getBooks().forEach(System.out::println);
        p.getSchoolBooks().forEach(System.out::println);
        p.getSpecialistBooks().forEach(System.out::println);
    });
}
```

```
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
```

```
SchoolBook [2, 2222, 2500.0, 5, English, RedLine]
SchoolBook [1, 1111, 2000.0, 5, Mathematics, Mathe5]
```

```
SpecialistBook [Computer, 3, 3333, 5000.0, Java]
```

Hier wird nur ein einziger `SELECT` zur Datenbank geschickt.

9 Versionierung und optimistische Sperren

Im folgenden geht's um Versionierung und um optimistische Sperren.

Im Gegensatz zu richtigen Sperren - den "pessimistischen Sperren" - sind optimistische Sperren überhaupt keine Sperren: sie heißen nur so.

Hibernate etwa rät von pessimistischen Sperren ab. Optimistische Sperren benötigen weniger Ressourcen und sind performanter. Die Hibernate-Dokumentation zum Thema "Optimistic concurrency control":

"The only approach that is consistent with high concurrency and high scalability is optimistic concurrency control with versioning. Version checking uses version numbers, or timestamps, to detect conflicting updates (and to prevent lost updates)."

JPA unterstützt automatische Versionierung und ermöglicht das automatische Erkennen von "conflicting updates".

9.1 Basics

In diesem Abschnitt wird zunächst gezeigt, wie JPA automatisch Versionsnummern verwaltet. (Das Erkennen von "conflicting updates" wird im folgenden Abschnitt gezeigt.)

In der `BOOK`-Tabelle wird eine Spalte für die Versionsnummer vorgesehen (`VERSION`):

create.sql

```
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    VERSION integer,  
    ISBN varchar (20) not null,  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    primary key (ID),  
    unique (ISBN)  
);
```

Die `Book`-Klasse wird mit einem `version`-Attribut ausgestattet (der Name des Attributs kann beliebig gewählt werden – per Konvention sollte aber `version` verwendet werden):

Book

```
package domain;  
// ...  
@Entity  
public class Book {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    @Version  
    private int version;  
  
    @Basic  
    private String isbn;  
  
    @Basic  
    private String title;  
  
    @Basic  
    private double price;  
  
    // Konstruktoren, getter, setter, toString...  
}
```

Man beachte, dass die `setVersion`-Methode geschützt ist. Das `version`-Attribut soll nur von JPA verwaltet werden– ein fremder Client soll die `version` nur lesen können (`getVersion` ist `public`).

Wichtig ist hier: das neue `version`-Attribut wird von der Klasse selbst nicht weiter verwaltet (hochgezählt etc.). Die Verwaltung dieses Attributs ist die ausschließliche Angelegenheit von JPA.

In der folgenden Applikation wird ein `Book` dreimal aktualisiert. Dementsprechend wird auch die Versionsnummer automatisch auf 3 inkrementiert:

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10));
    });
}
```

```
static void demoIncreasePrice(TransactionTemplate tt) {
    for (int i = 0; i < 3; i++) {
        tt.run(manager -> {
            final String s =
                "select b from Book b where b.isbn = :isbn";
            final TypedQuery<Book> query =
                manager.createQuery(s, Book.class);
            query.setParameter("isbn", "1111");
            final Book book = query.getSingleResult();
            System.out.println(book);
            book.setPrice(book.getPrice() + 0.50);
        });
    }
}
```

Die Ausgaben:

```
Hibernate: select ... from Book where isbn=?
Book [1, 1111, 10.0, Pascal, 0]
Hibernate: update Book set isbn=?, price=?, title=?, version=?
where id=? and version=?
Hibernate: select ... from Book where isbn=?
Book [1, 1111, 10.5, Pascal, 1]
Hibernate: update Book set isbn=?, price=?, title=?, version=?
where id=? and version=?
Hibernate: select ... from Book where isbn=?
Book [1, 1111, 11.0, Pascal, 2]
Hibernate: update Book set isbn=?, price=?, title=?, version=?
where id=? and version=?
```

Und hier der Zustand der Datenbank nach Ausführung der obigen Anwendung:

BOOK

ID	VERSION	ISBN	TITLE	PRICE
----	---------	------	-------	-------

1	3	1111	Pascal	11.5
---	---	------	--------	------

9.2 Optimistic Locking

Dieser Abschnitt zeigt eines der beiden Verfahren, wie "conflicting updates" automatisch erkannt und vermieden werden. Es wird dieselbe `Book`-Klasse und dieselbe `BOOK`-Tabelle verwendet wie im letzten Abschnitt.

Die Situation, welche die Testapplikation demonstriert, kann wie folgt beschrieben werden:

Es gibt mehrere Klienten, die "gleichzeitig" auf `Book`-Daten zugreifen und diese ändern wollen. Die Klienten sind GUI- oder WEB-Anwendungen. Ein Benutzer fordert zunächst ein `Book` zur Ansicht an. Dann denkt er eine Zeitlang nach ("user think time") und entschließt sich dann zu einer Änderung. In "derselben" Zeit hätte nun aber auch ein anderer nach dem gleichen Muster auf dasselbe `Book` zugreifen können. Und dann gibt es ein Problem.

Es ist klar, dass Transaktionen möglichst kurz sein müssen. Das heißt, dass eine Transaktionen auf keinen Fall während der Zeit zwischen dem Lese- und dem Update-Zugriff aktiv sein darf. Eine Möglichkeit besteht dann darin, einfach zwei Transaktionen zu verwenden – eine Transaktionen für den jeweiligen Lesezugriff eines Klienten, und eine andere für den Update-Zugriff. Die über den Lesezugriff bereitgestellte Tabellenzeile sollte natürlich auch nicht während der ganzen Zeit gesperrt sein ("pessimistic concurrency control"). In der zweiten Transaktionen muss also die Zeile erneut gelesen werden und dessen aktuelle Versionsnummer mit der Versionsnummer der in der ersten Transaktionen gelesenen Zeile verglichen werden. Wenn erstere größer ist als letztere, dann wurde die Zeile zwischendurch von einem anderen Klienten geändert. Wie in diesem Falle dann zu verfahren ist, ist natürlich spezifisch für die jeweilige Anwendung. Auf diese Weise kann aber auf jeden Fall ein zwischenzeitlicher Update erkannt werden – also das potentielle Problem des "lost updates" vermieden werden.

Die folgende Testanwendung simuliert den "gleichzeitigen" Zugriff zweier Clients mittels zweier `Threads`, die ein wenig zeitversetzt gestartet werden. Jeder der beiden `Threads` führt zwei aufeinanderfolgende Transaktionen (mit jeweils einem neuen `EntityManager`) aus: in der ersten Transaktion wird gelesen, in der zweiten findet ein Update statt. Zwischen den beiden Transaktionen wird die "user think time" mittels eines `Thread.sleep` simuliert.

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        manager.persist(new Book("1111", "Pascal", 10));  
    });  
}
```

```

static void demoConcurrentAccess(TransactionTemplate tt) {

    class Client extends Thread {
        @Override
        public void run() {
            try {
                final long id = Thread.currentThread().getId();
                final Book book = tt.runWithResult(manager -> {
                    System.out.println("read " + id);
                    final String jpql =
                        "select b from Book b " +
                        "where b.isbn = :isbn";
                    final TypedQuery<Book> query =
                        manager.createQuery(jpql, Book.class);
                    query.setParameter("isbn", "1111");
                    return query.getSingleResult();
                });
                Thread.sleep(2000);
                book.setPrice(book.getPrice() + 0.50);
                tt.run(manager -> {
                    System.out.println("update " + id);
                    manager.merge(book);
                });
            }
            catch (final InterruptedException e) {
                throw new RuntimeException(e);
            }
            catch (final OptimisticLockException e) {
                System.out.println("Expected Exception: " +
                    e.getMessage());
            }
        }
    }

    try {
        final Client client1 = new Client();
        final Client client2 = new Client();
        client1.start();
        Thread.sleep(1000);
        client2.start();
        client1.join();
        client2.join();
    }
    catch (final Exception e) {
        throw new RuntimeException(e);
    }
}

```

Der zuerst gestartete Client wird "gewinnen". Und wenn der zweite Client den Update versucht, wird eine Exception geworfen:

Hier die Ausgaben:

```

+-----
| demoConcurrentAccess
+-----

read 13
Hibernate: select ... Book where isbn=?
read 14
Hibernate: select ... Book where isbn=?
update 13
Hibernate: select ... Book where isbn=?
Hibernate: update Book set isbn=?, price=?, title=?, version=?
      where id=? and version=?
update 14
Hibernate: select ... Book where isbn=?
Expected Exception: Row was updated or deleted by another transaction
(or unsaved-value mapping was incorrect) : [domain.Book#1]

```

Und hier der Zustand der Datenbank:

```

BOOK
ID VERSION ISBN TITLE  PRICE
-----
1  1      1111 Pascal  10.5
-----

```

10 Stored Procedures

Stored Procedures werden in den verschiedenen Datenbanken unterschiedlich formuliert. Wir stellen im folgenden Beispiele vor, die mit HSQLDB funktionieren – mit Derby funktionieren sie nicht(!).

Im ersten der folgenden Beispiele lassen wir die Datenbank die Wurzel einer Zahl berechnen (ein nicht sehr intelligentes Beispiel...)

Im zweiten Beispiel zeigen wir, wie mittels `createNativeQuery` (also mittels SQL-Syntax) eine Stored Procedure aufgerufen werden kann, die eine Liste von `BookData`-Objekten zurückliefert.

Im dritten Beispiel benutzen wir zum selben Zweck die `EntityManager`-Methode `createStoredProcedureQuery` – benutzen also nur JPA-Mittel.

Im letzten Beispiel wird eine `NamedStoredProcedureQuery` verwendet.

10.1 Eine einfache Procedure

Die folgende HSQLDB-Stored Procedure berechnet und die Wurzel einer ihr übergebenen Zahl und liefert diese Wurzel zurück:

create.sql

```
CREATE FUNCTION SQRT(IN val DOUBLE)
  RETURNS DOUBLE
  LANGUAGE JAVA DETERMINISTIC NO SQL
  EXTERNAL NAME 'CLASSPATH:java.lang.Math.sqrt';
```

Application

Die Stored-Procedure kann mittels `createNativeQuery` wie folgt aufgerufen werden:

```
private static void demoSqrt(final TransactionTemplate tt) {
    tt.run(manager -> {
        Query query = manager.createNativeQuery("{call SQRT(?) }");
        query.setParameter(1, 49);
        double result = (double) query.getSingleResult();
        System.out.println("result: " + result);
    });
}
```

Die Ausgabe:

```
Hibernate: {call SQRT(?) }
result: 7.0
```

10.2 Native Queries

Im folgenden wird innerhalb einer Stored Procedure eine Menge von Büchern selektiert und diese in Form einer `List<BookData>` zurückgeliefert.

create.sql

```
create table BOOK (  
    ID integer generated by default as identity (start with 1),  
    ISBN varchar (20) not null,  
    TITLE varchar (128) not null,  
    PRICE double not null,  
    primary key (ID),  
    unique (ISBN)  
)
```

```
CREATE FUNCTION FindBooksByPrice (IN minPrice INTEGER, IN maxPrice INTEGER)  
RETURNS TABLE (id INTEGER, isbn VARCHAR(20), TITLE VARCHAR (128), PRICE  
DOUBLE)  
READS SQL DATA  
BEGIN ATOMIC  
    RETURN TABLE ( select id, isbn, title, price from BOOK alter  
        WHERE price >= minPrice AND price <= maxPrice);  
END
```

Book

```
package domain;  
// ...  
@Entity  
public class Book {  
  
    // Attribute id, isbn, title, price ...  
  
    // Konstruktoren, getter, setter, toString...  
}
```

BookData

```
package domain;  
  
public class BookData {  
  
    private String isbn;  
    private String title;  
    private double price;  
  
    // Konstruktoren, getter, setter, toString...  
}
```

Application

```
static void demoPersist(final TransactionTemplate tt) {  
    tt.run(manager -> {  
        manager.persist(new Book("1111", "Pascal", 10.0));  
        manager.persist(new Book("2222", "Modula", 20.0));  
        manager.persist(new Book("3333", "Oberon", 30.0));  
        manager.persist(new Book("4444", "Eiffel", 40.0));  
    });  
}
```

```
static void demoFind(final TransactionTemplate tt) {  
    tt.run(manager -> {  
        Query query = manager.createNativeQuery(  
            "{call FindBooksByPrice(?, ?)}");  
        query.setParameter(1, 20);  
        query.setParameter(2, 30);  
        createBookDataList(query).forEach(x -> System.out.println(x));  
    });  
}
```

```
private static List<BookData> createBookDataList(Query query) {  
    List<BookData> books = new ArrayList<>();  
    @SuppressWarnings("unchecked")  
    List<Object[]> rows = query.getResultList();  
    for (Object[] row : rows) {  
        // row[0] == id, not used  
        String isbn = (String) row[1];  
        String title = (String) row[2];  
        double price = (double) row[3];  
        books.add(new BookData(isbn, title, price));  
    }  
    return books;  
}
```

Die Ausgaben:

```
Hibernate: {call FindBooksByPrice(?, ?)}  
BookData [2222, 20.0, Modula]  
BookData [3333, 30.0, Oberon]
```


10.3 StoredProcedureQuery

Die folgende Anwendung leistet dasselbe wie die Anwendung des letzten Abschnitts. Allerdings benutzt sie statt `createNativeQuery` die Methode `createStoredProcedureQuery` (und ist somit nicht mehr von jeweiligen SQL-Spezialitäten abhängig):

```
private static void demoFind(final TransactionTemplate tt) {
    tt.run(manager -> {
        StoredProcedureQuery query =
            manager.createStoredProcedureQuery(
                "FindBooksByPrice");
        query.registerStoredProcedureParameter(
            "minPrice", Integer.class, ParameterMode.IN);
        query.registerStoredProcedureParameter(
            "maxPrice", Integer.class, ParameterMode.IN);

        query.setParameter("minPrice", 20);
        query.setParameter("maxPrice", 30);

        if (query.execute())
            createBookDataList(query)
                .forEach(x -> System.out.println(x));
        else
            System.out.println(
                "The query didn't return a resultset");
    });
}
```

Die Ausgaben:

```
Hibernate: {call FindBooksByPrice(?,?)}
BookData [2222, 20.0, Modula]
BookData [3333, 30.0, Oberon]
```

10.4 NamedStoredProcedureQuery

Die folgende Anwendung leistet dasselbe wie die Anwendungen der letzten beiden Abschnitte. Sie benutzt nun aber Named Stored Procedure Queries:

```
package domain;
// ...
import javax.persistence.NamedStoredProcedureQuery;
import javax.persistence.ParameterMode;
import javax.persistence.StoredProcedureParameter;

@Entity
@NamedStoredProcedureQuery (
    name          = "findBooksByPrice",
    procedureName  = "FindBooksByPrice",
    parameters = {
        @StoredProcedureParameter(mode = ParameterMode.IN,
                                   type = Integer.class, name = "minPrice"),
        @StoredProcedureParameter(mode = ParameterMode.IN,
                                   type = Integer.class, name = "maxPrice")
    }
)

public class Book {
    // ...
}

private static void demoFind(final TransactionTemplate tt) {
    tt.run(manager -> {
        StoredProcedureQuery query = manager
            .createNamedStoredProcedureQuery("findBooksByPrice");
        query.setParameter("minPrice", 20);
        query.setParameter("maxPrice", 30);

        if (query.execute())
            createBookDataList(query)
                .forEach(x -> System.out.println(x));
        else
            System.out.println(
                "The query didn't return a resultset");
    });
}
```

Die Ausgaben:

```
Hibernate: {call FindBooksByPrice(?,?)}
BookData [2222, 20.0, Modula]
BookData [3333, 30.0, Oberon]
```

11 Converter – Mapping von Spalten

Normalerweise kümmert sich Hibernate um das Mapping der Attribute / Properties von Objekten auf die Spalten der Tabellen. Java-Strings werden auf VARCHAR abgebildet, int wird auf INTEGER abgebildet etc.

Mitunter sind die Typen der Attribute / Properties der zu mappenden Java-Klassen aber benutzer-definierte Typen. Diese Typen kann Hibernate nicht kennen und daher auch kein automatisches Mapping unterstützen.

Die Aufgabe eines benutzerdefinierten Mappers (Converters) für benutzerdefinierte Typen besteht darin, zu entscheiden, wie ein Objekt des benutzerdefinierten Typs transformiert werden soll in einen Wert, der in einer Tabellenspalte gespeichert werden kann. Und umkehrt: zu entscheiden, welches Objekt des benutzerdefinierten Typs beim Lesezugriff aufgrund eines Wertes einer Tabellenspalte zurückzuliefern ist.

Eine Converter-Klasse muss das JPA-Interface `AttributeConverter<H,L>` implementieren (H steht für den "high-level" Typ, L steht für den "low-level"-Typ):

```
public interface AttributeConverter<H, L> {  
    public abstract L convertToDatabaseColumn(H value);  
    public abstract H convertToEntityAttribute(Lvalue);  
}
```

11.1 Expizites Mapping

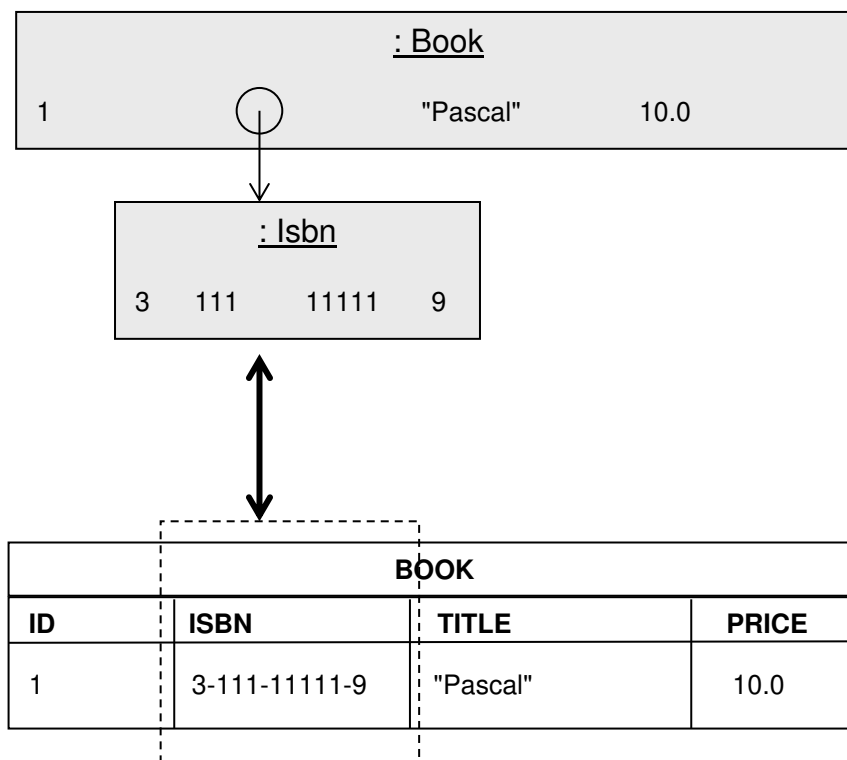
Eine ISBN-Nummer enthält vier Elemente: Land, Verlag, Verlags-Nummer und ein Check-Character. Wir definieren eine `Isbn`-Klasse, welche für jedes dieser Elemente ein Attribut definiert.

In der Datenbank wird die ISBN-Nummer wie gehabt als `VARCHAR` gespeichert:

create.sql

```
CREATE TABLE BOOK (
    // ...
    ISBN VARCHAR(20),
    // ...
);
```

Ein Objekt-Diagramm:



Ein `VARCHAR` muss auf ein `Isbn`-Objekt gemappt werden und umgekehrt.

Hier zunächst die Klasse `Isbn`:

Isbn

```
package domain;

final public class Isbn {

    private final String language;
    private final String publisherNumber;
    private final String bookNumber;
    private final String check;

    public Isbn(String language, String publisherNumber, String bookNumber,
        String check) {
        this.language = language;
        this.publisherNumber = publisherNumber;
        this.bookNumber = bookNumber;
        this.check = check;
        if (this.getStringValue().length() != 13)
            throw new IllegalArgumentException();
    }

    public Isbn(String isbn) {
        final String[] tokens = isbn.split("-");
        this.language = tokens[0];
        this.publisherNumber = tokens[1];
        this.bookNumber = tokens[2];
        this.check = tokens[3];
        if (this.getStringValue().length() != 13)
            throw new IllegalArgumentException();
    }

    public String getLanguage() {
        return this.language;
    }
    public String getPublisherNumber() {
        return this.publisherNumber;
    }
    public String getBookNumber() {
        return this.bookNumber;
    }
    public String getCheck() {
        return this.check;
    }

    public String getStringValue() {
        return this.language + "-" + this.publisherNumber + "-"
            + this.bookNumber + "-" + this.check;
    }

    @Override public boolean equals(Object other) { ... }
    @Override public int hashCode() { ... }

    @Override public String toString() { ... }
}
```

Man beachte, dass es sich bei dieser Klasse um keine Bean-Klasse handelt (es fehlt der parameterlose Konstruktor). Im Sinne von Hibernate ist `Isbn` also auch keine persistente Klasse. Daher existiert auch keine `@Entity`-Annotation. Es handelt sich also um eine einfache Hilfsklasse (die aber serialisierbar sein sollte).

Objekte dieser Klasse sind konstant – einmal erzeugt, kann ihr Zustand anschließend nicht mehr geändert werden (dies ist für die folgenden Überlegungen wichtig!). Die Klasse kann auch nicht als Basisklasse für weitere Ableitungen verwendet werden (sie ist `final`).

Dem ersten Konstruktor werden die einzelnen Bestandteile einer ISBN-Nummer übergeben; dem zweiten eine "komplette" ISBN-Nummer (in "lesbarer" Form). Der zweite Konstruktor muss daher den ihm übergebenen String in seine Bestandteile zerlegen.

Die Methode `getStringValue` gibt die ISBN-Nummer in "lesbarer" Form zurück. Man beachte auch, dass `equals` und `hashCode` überschrieben sind (diese Eigenschaft wird von der Klasse `Book` genutzt werden).

Book

Das Attribut `isbn` hat neben der `@Basic`-Annotation nun zusätzlich eine `@Convert`-Annotation, welche den zu verwendenden Converter definiert:

```
package domain;
// ...
import javax.persistence.Convert;

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    @Convert(converter=IsbnConverter.class)
    private Isbn isbn;

    @Basic
    private String title;

    Book() { }
    public Book(Isbn isbn, String title) { ... }

    // getter, setter, toString ...
}
```

IsbnConverter

```
package domain;

import javax.persistence.AttributeConverter;

public class IsbnConverter implements AttributeConverter<Isbn, String> {
    @Override
    public String convertToDatabaseColumn(Isbn isbn) {
        return isbn == null ? null : isbn.getStringValue();
    }
    @Override
    public Isbn convertToEntityAttribute(String value) {
        return value == null ? null : new Isbn(value);
    }
}
```

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book(
            new Isbn("3", "111", "11111", "9"), "Pascal"));
        manager.persist(new Book(
            new Isbn("3", "222", "22222", "8"), "Modula"));
    });
}
```

Der Zustand der Datenbank nach Ausführung der Persist-Transaktion:

```
BOOK
ID ISBN          TITLE
-----
1  3-111-11111-9 Pascal
2  3-222-22222-8 Modula
-----
```

```
static void demoQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b where b.isbn = :isbn";
        final TypedQuery<Book> query = manager.createQuery(
            jpql, Book.class);
        query.setParameter("isbn", new Isbn("3", "222", "22222", "8"));
        query.getResultList().forEach(System.out::println);
    });
}
```

Die Ausgaben:

```
Book [Isbn [3-222-22222-8], Modula]
```

11.2 Auto Apply

Statt jedes Attribut einer jeden Entity-Klasse, die einen Converter verwendet, mit `@Convert` auszustatten, kann eine Converter-Klasse ihrerseits mit der `@Converter`-Annotation ausgestattet werden. Diese Annotation hat ein Attribut `autoApply`, welches auf `true` gesetzt wird:

IsbnConverter

```
@Converter(autoApply = true)
public class IsbnConverter implements AttributeConverter<Isbn, String> {
    // ...
}
```

Dann erübrigt sich in der `Book`-Klasse die Angabe der `@Convert`-Annotation:

Book

```
package domain;
// ...
@Entity
public class Book {

    @Basic
    private Isbn isbn;

    //
}
```


11.3 Enums

Enums können auf zweifache Weise konvertiert werden: zu einem String oder zu einer Ganzzahl. Wird die erste Variante genutzt, wird das Resultat des `name()`-Aufrufs auf die enum-Konstante genutzt; bei der zweiten Variante wird das Resultat des `ordinal()`-Aufrufs genutzt.

create.sql

Der Typ einen `BOOKS` soll als `VARCHAR` gespeichert werden:

```
CREATE TABLE BOOK (
    // ...
    BOOKTYPE VARCHAR(20),
    // ...
);
```

BookType

In Java ist der `BookType` als enum definiert:

```
package domain;

public enum BookType {
    SCHOOL_BOOK, SPECIALIST_BOOK
}
```

Book

Ein `Book` hat einen `BookType` - das entsprechende Attribut ist mit `@Enumerated` ausgezeichnet. Die Annotation hat das `value`-Attribut `EnumType.STRING` – wird ein enum-Wert gespeichert, wird als das Resultat des Aufrufs der `name()`-Methode verwendet:

```
package domain;
// ...
@Entity
public class Book {

    // ...

    @Enumerated(EnumType.STRING)
    private BookType bookType;

    // Konstruktoren, setter, getter, toString ...
}
```

Alternativ zu `EnumType.STRING` hätten wir `EnumType.ORDINAL` verwenden können. Dann wäre das Resultat des `ordinal()`-Aufrufs in der Datenbank gespeichert worden (dann müsste die entsprechende Tabellenspalte natürlich auch als `INTEGER` definiert werden).

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book(
            "1111", "Pascal", BookType.SPECIALIST_BOOK));
        manager.persist(new Book(
            "2222", "Modula", null));
        manager.persist(new Book(
            "9999", "Lerning English", BookType.SCHOOL_BOOK));
    });
}
```

Der Inhalt der BOOK-Tabelle:

BOOK	ID	ISBN	TITLE	BOOKTYPE
1	1111	Pascal	SPECIALIST_BOOK	
2	2222	Modula	NULL	
3	9999	Lerning English	SCHOOL_BOOK	

```
static void demoQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b where b.bookType = :bookType";
        final TypedQuery<Book> query = manager.createQuery(
            jpql, Book.class);
        query.setParameter("bookType", BookType.SPECIALIST_BOOK);
        query.getResultList().forEach(System.out::println);
    });
}
```

Die Ausgabe:

Book [1111, Pascal, SPECIALIST_BOOK]

11.4 Calendar

Die Tabelle `BOOK` enthalte eine weitere Spalte namens `PDATE` vom Typ `TIMESTAMP`:

create.sql

```
CREATE TABLE BOOK (  
    // ...  
    PDATE TIMESTAMP,  
    // ...  
);
```

Das der `PDATE`-Spalte entsprechende Attribut der Java-Klasse kann mit `@Temporal` gekennzeichnet werden. An `@Temporal` kann einer der folgenden `TemporalType`-enum-Werte übergeben werden: `DATE`, `TIME` oder `TIMESTAMP`.

Book

Wir verwenden ein Attribut vom Typ `Calendar` und eine `@Temporal`-Annotation mit dem Wert `TemporalType.DATE`:

```
package domain;  
// ...  
import java.util.Calendar;  
import javax.persistence.Temporal;  
import javax.persistence.TemporalType;  
  
@Entity  
public class Book  
    // ...  
  
    @Basic  
    @Column(name = "PDATE")  
    @Temporal(TemporalType.DATE)  
    // @Temporal(TemporalType.TIMESTAMP)  
    private Calendar date;  
  
    // ...  
}
```

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final Calendar c1 = Calendar.getInstance();  
        c1.set(1970, 1, 1, 13, 59, 59);  
        manager.persist(new Book("1111", "Pascal", c1));  
        final Calendar c2 = Calendar.getInstance();  
        c2.set(1980, 12, 31, 23, 59, 59);
```

```

        manager.persist(new Book("2222", "Modula", c2));
    });
}

```

Die Datenbank nach Ausführung der Persist-Transaktion:

```

BOOK
ID ISBN TITLE  PDATE
-----
1  1111 Pascal 1970-02-01 00:00:00.0
2  2222 Modula 1981-01-31 00:00:00.0
-----

```

```

static void demoQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select b from Book b where b.date = :date";
        final TypedQuery<Book> query =
            manager.createQuery(jpql, Book.class);
        final Calendar c = Calendar.getInstance();
        c.set(1970, 1, 1, 13, 59, 59);
        query.setParameter("date", c);
        query.getResultList().forEach(System.out::println);
    });
}

```

Die Ausgaben:

```

Book [1111, Pascal, GregorianCalendar ... 1970 1 1 13 59 59 ...

```

11.5 LocalDate

Wir verwenden dieselbe Tabellenstruktur wie im letzten Abschnitt (eine Spalte namens PDATE vom Typ TIMESTAMP).

Create Table

```
CREATE TABLE BOOK (
    // ...
    PDATE TIMESPTAMP,
    // ...
);
```

Book

Das date-Attribut von Book ist nun aber vom Typ LocalDate:

```
package domain;
// ...
import java.time.LocalDate;

@Entity
public class Book {
    // ...
    @Basic
    @Column(name = "PDATE")
    private LocalDate date;
    // ...
}
```

Hier darf keine @Temporal-Annotation verwendet werden!

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal",
            LocalDate.of(1970, Month.JANUARY, 1)));
        manager.persist(new Book("2222", "Modula",
            LocalDate.of(1980, Month.DECEMBER, 31)));
    });
}
```

Die Tabelle nach Ausführung der obigen Transaktion:

```
BOOK
ID ISBN TITLE  DATE
-----
1  1111 Pascal 1970-01-01
```

2 2222 Modula 1980-12-31

```
static void demoQuery(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final String jpql =  
            "select b from Book b where b.date = :date";  
        final TypedQuery<Book> query =  
            manager.createQuery(jpql, Book.class);  
        query.setParameter("date",  
            LocalDate.of(1980, Month.DECEMBER, 31));  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

Die Ausgabe:

Book [2222, Modula, 1980-12-31]

12Criteria

JPA stellt neben JPQL eine weitere Möglichkeit zur Verfügung, Abfragen zu spezifizieren: Criteria. Während die Verwendung von JPQL die Kenntnis der Syntax einer Sprache (eben JPQL) voraussetzt, werden Criteria vollständig in Java spezifiziert. Die Kenntnis von JPQL ist also nicht mehr erforderlich. Bei der Verwendung von Criteria sind somit auch bestimmte Fehler in der Formulierung von JPQL-Strings ausgeschlossen – Fehler, die natürlich erst zur Laufzeit erkannt werden können. Mittels des Criteria-Konzepts können typischere Abfragen formuliert werden.

12.1 Einfache Queries

Im folgenden wird eine Anwendung vorgestellt, welche aus einer Persist- und mehreren Query-Transaktionen besteht.

Application

```
static void demoPersist(TransactionTemplate tt) {  
    tt.run(manager -> {  
        manager.persist(new Book("1111", "Pascal", 30.0));  
        manager.persist(new Book("2222", "Modula", 30.0));  
        manager.persist(new Book("3333", "Oberon", 20.0));  
        manager.persist(new Book("4444", "Eiffel", 60.0));  
        manager.persist(new Book("5555", "C", 10.0));  
        manager.persist(new Book("6666", "C++", 40.0));  
        manager.persist(new Book("7777", "C#", 50.0));  
    });  
}
```

Die erste Query-Transaktion ermittelt alle Books:

```
static void demoQuery1(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final CriteriaBuilder builder =  
            manager.getCriteriaBuilder();  
        final CriteriaQuery cq = builder.createQuery(Book.class);  
        final Root<Book> root = cq.from(Book.class);  
        cq.select(root);  
        final TypedQuery<Book> query = manager.createQuery(cq);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```

Die zweite Transaktion ermittelt das Book mit der Isbn "1111":

```
static void demoQuery2(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final CriteriaBuilder builder =  
            manager.getCriteriaBuilder();  
        final CriteriaQuery cq = builder.createQuery(Book.class);  
        final Root<Book> root = cq.from(Book.class);  
        final EntityType<Book> et = root.getModel();  
        final SingularAttribute a =  
            et.getSingularAttribute("isbn");  
        final Expression e = builder.equal(root.get(a), "1111");  
        cq.where(e);  
        cq.select(root);  
        final TypedQuery<Book> query = manager.createQuery(cq);  
        query.getResultList().forEach(System.out::println);  
    });  
}
```


Die folgende Zeile ist hier natürlich problematisch:

```
final SingularAttribute a = et.getSingularAttribute("isbn");
```

Der Compiler kann nicht garantieren, dass die Klasse `Book` das Attribut `"isbn"` definiert...

Die obige Transaktion hätte man auch wie folgt formulieren können:

```
static void demoQuery3(TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery cq = builder.createQuery(Book.class);
        final Root<Book> root = cq.from(Book.class);
        final Expression e =
            builder.equal(root.get("isbn"), "1111");
        cq.where(e);
        cq.select(root);
        final TypedQuery<Book> query = manager.createQuery(cq);
        query.getResultList().forEach(System.out::println);
    });
}
```

Die folgende Transaktion ermittelt alle Bücher, deren Preis zwischen 20.0 und 30.0 liegt:

```
static void demoQuery4(TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery cq = builder.createQuery(Book.class);
        final Root<Book> root = cq.from(Book.class);
        final Expression e =
            builder.between(root.get("price"), 20.0, 30.0);
        cq.where(e);
        cq.select(root);
        final TypedQuery<Book> query = manager.createQuery(cq);
        query.getResultList().forEach(System.out::println);
    });
}
```

Die folgende Transaktion ermittelt alle Bücher, deren Titel mit "C" beginnt:

```
static void demoQuery5(TransactionTemplate tt) {
    Util.mlog();
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery cq = builder.createQuery(Book.class);
        final Root<Book> root = cq.from(Book.class);
        final Expression e =
            builder.like(root.get("title"), "C%");
    });
}
```

```

        cq.where(e);
        cq.select(root);
        final TypedQuery<Book> query = manager.createQuery(cq);
        query.getResultList().forEach(System.out::println);
    });
}

```

Die nächste Transaktion ermittelt alle Bücher, deren Preis zwischen 20.0 und 30.0 liegt und deren Titel mit "C" beginnt:

```

static void demoQuery6(TransactionTemplate tt) {
    Util.mlog();
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery cq = builder.createQuery(Book.class);
        final Root<Book> root = cq.from(Book.class);
        final Expression e1 =
            builder.like(root.get("title"), "C%");
        final Expression e2 =
            builder.between(root.get("price"), 10.0, 40.0);
        final Expression e =
            builder.and(e1, e2);
        cq.where(e);
        cq.select(root);
        final TypedQuery<Book> query = manager.createQuery(cq);
        query.getResultList().forEach(System.out::println);
    });
}

```

Die nächste Transaktion verknüpft die beiden "like" und "between"-Expressions mittels OR (was natürlich nicht allzu sinnvoll ist...):

```

static void demoQuery7(TransactionTemplate tt) {
    Util.mlog();
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery cq = builder.createQuery(Book.class);
        final Root<Book> root = cq.from(Book.class);
        final Expression e1 =
            builder.like(root.get("title"), "C%");
        final Expression e2 =
            builder.between(root.get("price"), 10.0, 40.0);
        final Expression e =
            builder.or(e1, e2);
        cq.where(e);
        cq.select(root);
        final TypedQuery<Book> query = manager.createQuery(cq);
        query.getResultList().forEach(System.out::println);
    });
}

```

Die obigen Beispiel stellen nur die Spitze des Criteria-Eisbergs vor.

Die Verwendung von Criteria anstelle von JPQL ist sicherlich gewöhnungsbedürftig. Aber das Verfahren ist insbesondere dann interessant, wenn Abfragen zur Laufzeit erstellt werden müssen – z.B. aufgrund von Benutzereingaben. Anstelle JPQL-Strings "zusammenzubasteln", ist es dann sinnvoll, Criteria-Objekte zusammenzubauen. Den korrekten Zusammenbau solcher Objekte kann der Compiler überprüfen.

In den obigen Beispielen wurden die Namen der Attribute / Properties als String-Literale übergeben. Hier existiert eine sicherere Variante. Aufgrund des Metamodels von JPA kann man einen Generator schreiben, welcher aufgrund automatisch z.B. die folgende Klasse generiert:

Book_

```
package domain;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;
@StaticMetamodel(Book.class)
public abstract class Book_ {
    public static volatile SingularAttribute<Book, Integer> id;
    public static volatile SingularAttribute<Book, String> title;
    public static volatile SingularAttribute<Book, Double> price;
    public static volatile SingularAttribute<Book, String> isbn;
}
```

Die Attributes der persistenten Book-Klasse könnten dann über die in Book_ definierten Konstanten typsicher angesprochen werden.

Wollte man auf diesem Wege (über das Metamodel) die Book_-Klasse generieren, müsste die Book-Klasse natürlich bereits übersetzt sein. Der build würde also aus zwei Phasen bestehen. Eine andere Möglichkeit besteht darin, das in Java 6 definierte Annotation-Processor-Konzept zu nutzen. Ohne hier auf die Einzelheiten dieses Konzepts einzugehen: der zweistufige Build entfällt hier.

Hibernate hat einen solchen Annotation-Processor implementiert. Er befindet sich in folgender jar-Datei:

```
hibernate-jpamodelgen-4.3.8.Final.jar
```

Diese Klasse muss unter Project > Properties > Java Compiler > Annotation Processing > FactoryPath eingetragen werden.

Das Result besteht dann darin, dass die Book_-Klasse (Book_.java) automatisch im Zuge der normalen Compilation erzeugt wird (und dann natürlich auch compiliert wird).

12.2 Komplexe Queries

Im folgenden werden einige komplexere Queries vorgestellt, die mit Criteria implementiert sind. Es geht u.a. darum, einen join-fetch zu veranlassen.

Die Datenbank umfasst vier Tabellen (diese werden direkt in der `create.sql` gefüllt):

create.sql

```
create table PUBLISHER (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table AUTHOR (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PRICE double not null,
    PUBLISHER_ID integer,
    primary key (ID),
    unique (ISBN),
    foreign key (PUBLISHER_ID) references PUBLISHER
);

create table BOOK_AUTHOR (
    BOOKS_ID integer,
    AUTHORS_ID integer,
    primary key (BOOKS_ID, AUTHORS_ID),
    foreign key (BOOKS_ID) references BOOK,
    foreign key (AUTHORS_ID) references AUTHOR
);

// weitere INSERT-Statements ...
```

Der Zustand der hier erzeugten Datenbank:

```
AUTHOR
ID NAME
-----
1  Wirth
2  Meyer
```

```

-----
BOOK
ID ISBN TITLE PRICE PUBLISHER_ID
-----
1  1111 Pascal 10.0  1
2  2222 Modula 20.0  1
3  3333 Oberon 30.0  2
-----

```

```

BOOK_AUTHOR
BOOKS_ID AUTHORS_ID
-----
1          1
2          1
2          2
3          1
3          2
-----

```

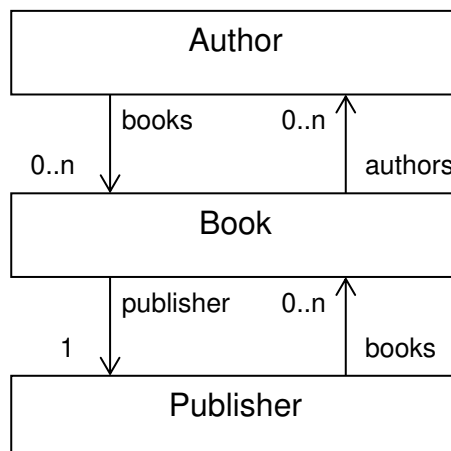
```

PUBLISHER
ID NAME
-----
1  Addison
2  Prentice
-----

```

Klassendiagramm

Wir benutzen drei Entity-Klassen: `Author`, `Book` und `Publisher`. Es existiert eine M-N-Verbindung zwischen `Author` und `Book`; und eine bidirektionale 1:N-Beziehung zwischen `Publisher` und `Book`:



Author

```
package domain;
// ...
@Entity
public class Author {
    // Attribute id und name ...

    @ManyToMany(mappedBy = "authors")
    private Set<Book> books = new HashSet<Book>();

    // Konstruktoren, setter, getter, toString ...
}
```

Publisher

```
package domain;
// ...
@Entity
public class Publisher {
    // Attribute id und name ...

    @OneToMany(mappedBy = "publisher")
    private Set<Book> books = new HashSet<Book>();

    // Konstruktoren, setter, getter, toString ...
}
```

Book

```
package domain;
// ...
@Entity
public class Book {
    // Attribute id, isbn, title, price ...

    @ManyToOne(fetch=FetchType.LAZY)
    private Publisher publisher;

    @ManyToMany(fetch=FetchType.LAZY)
    private Set<Author> authors = new HashSet<Author>();

    // Konstruktoren, setter, getter, toString ...
}
```

BookData

Zusätzlich gibt's für Constructor-Queries eine BookData-Klasse:

```
package appl;
```

```

public class BookData {

    public String isbn;
    public String title;
    public Double price;

    public BookData(String isbn, String title, Double price) { ... }
    public BookData(String isbn, String title) { ... }

    // toString...
}

```

Application

```

static void demoFetch(TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery<Book> criteria =
            builder.createQuery(Book.class);
        final Root<Book> bookRoot = criteria.from(Book.class);
        bookRoot.fetch("publisher", JoinType.LEFT);
        criteria.select(bookRoot);

        final TypedQuery<Book> query =
            manager.createQuery(criteria);
        query.getResultList().forEach(b -> {
            System.out.println(b);
            System.out.println("\t" + b.getPublisher());
        });
    });
}

```

```

Hibernate: select ... from Book b
left outer join Publisher p on b.publisher_id=p.id
Book [1, 1111, 10.0, Pascal]
    Publisher [1, Addison]
Book [2, 2222, 20.0, Modula]
    Publisher [1, Addison]
Book [3, 3333, 30.0, Oberon]
    Publisher [2, Prentice]

```

```

static void demoFetchFetch(TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery<Book> criteria =
            builder.createQuery(Book.class);
        final Root<Book> bookRoot = criteria.from(Book.class);
        bookRoot.fetch("publisher", JoinType.LEFT);
        bookRoot.fetch("authors", JoinType.LEFT);
        criteria.select(bookRoot).distinct(true);

        final TypedQuery<Book> query =

```

```

        manager.createQuery(criteria);
        query.getResultList().forEach(b -> {
            System.out.println(b);
            System.out.println("\t" + b.getPublisher());
            b.getAuthors().forEach(a -> System.out.println("\t" + a));
        });
    });
}

```

Hibernate: select ...

```

Book [1, 1111, 10.0, Pascal]
    Publisher [1, Addison]
    Author [1, Wirth]
Book [2, 2222, 20.0, Modula]
    Publisher [1, Addison]
    Author [2, Meyer]
    Author [1, Wirth]
Book [3, 3333, 30.0, Oberon]
    Publisher [2, Prentice]
    Author [2, Meyer]
    Author [1, Wirth]

```

```

static void demoQueryFetchWhere(TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery<Book> criteria =
            builder.createQuery(Book.class);
        final Root<Book> bookRoot = criteria.from(Book.class);
        bookRoot.fetch("publisher", JoinType.LEFT);
        final Expression<Boolean> e = builder.equal(
            bookRoot.get("publisher").get("name"), "Addison");
        criteria.select(bookRoot).where(e);
        final TypedQuery<Book> query =
            manager.createQuery(criteria);
        query.getResultList().forEach(b -> {
            System.out.println(b);
            System.out.println("\t" + b.getPublisher());
        });
    });
}

```

Hibernate: select ...

```

Book [1, 1111, 10.0, Pascal]
    Publisher [1, Addison]
Book [2, 2222, 20.0, Modula]
    Publisher [1, Addison]

```

```

static void demoConstructorQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery<BookData> criteria =
            builder.createQuery(BookData.class);
    });
}

```



```

        final Root<Book> bookRoot = criteria.from(Book.class);
        criteria.select(builder.construct(
            BookData.class,
            bookRoot.get("isbn"),
            bookRoot.get("title"),
            bookRoot.get("price")));
        final TypedQuery<BookData> query1 =
            manager.createQuery(criteria);
        query1.getResultList().forEach(System.out::println);

        criteria.select(builder.construct(
            BookData.class,
            bookRoot.get("isbn"),
            bookRoot.get("title")));
        final TypedQuery<BookData> query2 =
            manager.createQuery(criteria);
        query2.getResultList().forEach(System.out::println);
    });
}

```

Hibernate: select ...

BookData [isbn=1111, title=Pascal, price=10.0]

BookData [isbn=2222, title=Modula, price=20.0]

BookData [isbn=3333, title=Oberon, price=30.0]

Hibernate: select ...

BookData [isbn=1111, title=Pascal, price=null]

BookData [isbn=2222, title=Modula, price=null]

BookData [isbn=3333, title=Oberon, price=null]

```

static void demoColumnProjectionQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery<Object[]> criteria =
            builder.createQuery(Object[].class);
        final Root<Book> bookRoot = criteria.from(Book.class);
        criteria.select(builder.construct(
            Object[].class,
            bookRoot.get("isbn"),
            bookRoot.get("title"),
            bookRoot.get("price")));

        final TypedQuery<Object[]> query1 =
            manager.createQuery(criteria);
        query1.getResultList().forEach(row -> {
            for(Object value : row)
                System.out.print(value + " ");
            System.out.println();
        });

        criteria.select(builder.construct(
            Object[].class,
            bookRoot.get("isbn"),
            bookRoot.get("title")));
        final TypedQuery<Object[]> query2 =

```

```

        manager.createQuery(criteria);
        query2.getResultList().forEach(row -> {
            for(Object value : row)
                System.out.print(value + " ");
            System.out.println();
        });
    });
}

```

```

Hibernate: select ...
1111 Pascal 10.0
2222 Modula 20.0
3333 Oberon 30.0
Hibernate: select ...
1111 Pascal
2222 Modula
3333 Oberon

```

```

static void demoSingleColumnProjectionQuery(
    TransactionTemplate tt) {
    tt.run(manager -> {
        final CriteriaBuilder builder =
            manager.getCriteriaBuilder();
        final CriteriaQuery<String> criteria =
            builder.createQuery(String.class);
        final Root<Book> bookRoot = criteria.from(Book.class);

        criteria.select(builder.construct(
            String.class,
            bookRoot.get("isbn")));
        final TypedQuery<String> query1 =
            manager.createQuery(criteria);
        query1.getResultList().forEach(System.out::println);

        criteria.select(builder.construct(
            String.class,
            bookRoot.get("title")));
        final TypedQuery<String> query2 =
            manager.createQuery(criteria);
        query2.getResultList().forEach(System.out::println);
    });
}

```

```

Hibernate: select ...
1111
2222
3333
Hibernate: select ...
Pascal
Modula
Oberon

```

12.3 Update / Delete

Mittels Criteria können auch Mengen-Updates und –Deletes ausgeführt werden. Zur Demonstration verwenden wir wieder die bekannte `BOOK`-Klasse.

Book

```
package domain;
// ...
@Entity
public class Book implements {

    // Attribute id, isbn, title, price

    // Konstruktoren, getter, setter, toString ...
}
```

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10.0));
        manager.persist(new Book("2222", "Modula", 20.0));
        manager.persist(new Book("3333", "Oberon", 30.0));
        manager.persist(new Book("4444", "Eiffel", 40.0));
    });
}
```

Die folgende Transaktion setzt die Preise aller Bücher, deren aktueller Preis größer als 25 ist, auf 99:

```
static void demoUpdate(TransactionTemplate tt) {
    tt.run(manager -> {
        CriteriaBuilder builder = manager.getCriteriaBuilder();
        CriteriaUpdate<Book> update =
            builder.createCriteriaUpdate(Book.class);
        Root<Book> root = update.from(Book.class);
        update.set("price", 99.0);
        update.where(builder.greaterThan(root.get("price"), 25.0));
        manager.createQuery(update).executeUpdate();
    });
}
```

Hibernate: update Book set price=99.0 where price>25.0

Die folgende Transaktion löscht alle Bücher, deren Preise kleiner als 25 sind:

```
static void demoDelete(TransactionTemplate tt) {
    tt.run(manager -> {
        CriteriaBuilder builder = manager.getCriteriaBuilder();
```

```
CriteriaDelete<Book> delete =
    builder.createCriteriaDelete(Book.class);
Root<Book> root = delete.from(Book.class);
delete.where(builder.lessThan(root.get("price"), 25.0));
manager.createQuery(delete).executeUpdate();
});
}
```

Hibernate: delete from Book where price<25.0

Der Zustand der BOOK-Tabelle nach Ausführung der obigen Methoden:

```
BOOK
ID ISBN TITLE  PRICE
-----
3  3333 Oberon 99.0
4  4444 Eiffel 99.0
-----
```

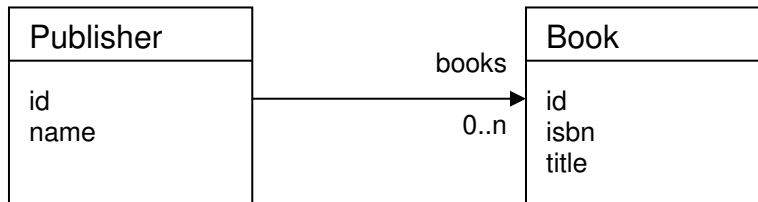
13 Entity-Graphen

Entity-Graphen sind in der Version 2.1 neu eingeführt worden.

Solche Graphen ermöglichen die Formulierung von Selects (entweder via JPQL oder Criteria), ohne dabei bereits das Fetch-Verhalten zu spezifizieren. Das Fetch-Verhalten – also die Spezifikation dessen, was als "Objekt-Wolke" zurückzuliefern ist – wird dann anschließend über solche Graphen festgelegt.

13.1 Explizite Erzeugung

Ein Publisher habe N Books:



create.sql

```

create table PUBLISHER (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PUBLISHER_ID integer,
    primary key (ID),
    unique (ISBN),
    foreign key (PUBLISHER_ID) references PUBLISHER
);
  
```

Book

```

package domain;
// ...
@Entity
public class Book {

    // Attribute id, isbn, title ...

    // Konstruktoren, getter, setter, toString...
}
  
```

Publisher

```

package domain;
// ...
@Entity
public class Publisher {
  
```

```
// Attribute id, name ...

@OneToMany(fetch=FetchType.LAZY)
@JoinColumn(name="PUBLISHER_ID") // required!!!
private Set<Book> books = new HashSet<Book>();

// Konstruktoren, getter, setter, toString...
}
```

Application

Zunächst werden zwei Publisher und vier Books persistiert. Jedem Publisher gehören zwei der vier Books:

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", p1));
        manager.persist(new Book("2222", "Modula", p1));
        manager.persist(new Book("3333", "Oberon", p1));
        manager.persist(new Book("4444", "Eiffel", p2));
    });
}
```

Die `demoPersist`-Methode persistiert zwei Publisher, wobei der erste mit drei Books und der zweite mit einem Book assoziiert ist.

```
static void demoStandardQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select p from Publisher p where name = 'Addison'";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        final Publisher p = query.getSingleResult();
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}
```

Im JPQL-String findet sich kein `fetch` – und wir benutzen noch keinen Entity Graphen. Wir selektieren den ersten der beiden Publisher und geben diesen zusammen mit seinen Books aus:

Die Ausgaben zeigen, dass die Books des Publishers via lazy loading nachgeladen werden:

```
Hibernate: select ... from Publisher ...
```

Publisher [1, Addison]

Hibernate: select ... from Book where ...

Book [1, 1111, Pascal]

Book [3, 3333, Oberon]

Book [2, 2222, Modula]

```
static void demoJoinFetchQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select p from Publisher p join fetch p.books " +
            "where name = 'Addison' ";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        final Publisher p = query.getSingleResult();
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}
```

Hier wird nun der `fetch` genutzt. Alle Daten werden mit einem einzigen `SELECT` geladen:

Hibernate: select ... from Publisher ... inner join Book ...

Publisher [1, Addison]

Book [1, 1111, Pascal]

Book [3, 3333, Oberon]

Book [2, 2222, Modula]

```
static void demoGraphQuery(TransactionTemplate tt) {
    tt.run(manager -> {
        final String jpql =
            "select p from Publisher p where name = 'Addison'";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);

        final EntityGraph<Publisher> graph =
            manager.createEntityGraph(Publisher.class);
        graph.addAttributeNodes("books");
        query.setHint("javax.persistence.fetchgraph", graph);

        final Publisher p = query.getSingleResult();
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}
```

Im JPQL-String fehlt nun wieder die `fetch`-Angabe.

Stattdessen existiert nun ein `EntityGraph<Publisher>`. Via `addAttributeNodes` wird diesem ein Knoten hinzugefügt, der das `books`-Attribut von `Publisher` referenziert. (An die Methode `addAttributeNodes` können beliebig viele Knoten übergeben werden – deshalb der Plural).

Der `EntityGraph` wird via `setHint` bei der `TypedQuery` registriert.

Auch hier wird nur ein einziger `SELECT` ausgeführt:

```
Hibernate: select ... from Publisher ... left outer join Book ...
Publisher [1, Addison]
    Book [1, 1111, Pascal]
    Book [3, 3333, Oberon]
    Book [2, 2222, Modula]
```

Wir können nun dieselbe "logischen" Abfrage mit verschiedenen Fetch-Strategien ausstatten.

```
static void demoGraphFind(TransactionTemplate tt) {
    tt.run(manager -> {
        final EntityGraph<Publisher> graph =
            manager.createEntityGraph(Publisher.class);
        graph.addAttributeNodes("books");
        final Map<String, Object> hints =
            new HashMap<String, Object>();
        hints.put("javax.persistence.fetchgraph", graph);

        final Publisher p = manager.find(Publisher.class, 1, hints);
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}
```

Soll statt der `createQuery`-Methode die `find`-Methode mit einem Entity-Graphen ausgestattet werden, muss dieser zunächst in eine `Map` eingetragen werden – eine `Map`, die dann an die `find`-Methode übergeben wird.

Die Ausgaben:

```
Hibernate: select ... from Publisher ... left outer join Book ...
Publisher [1, Addison]
    Book [1, 1111, Pascal]
    Book [3, 3333, Oberon]
    Book [2, 2222, Modula]
```

```
static void demoAddGraphToFactory(TransactionTemplate tt) {
    tt.run(manager -> {
        final EntityGraph<Publisher> graph =
            manager.createEntityGraph(Publisher.class);
        graph.addAttributeNodes("books");
        manager.getEntityManagerFactory().addNamedEntityGraph(
            "publisher.books", graph);
    });
    tt.run(manager -> {
        final EntityGraph<?> graph =
```

```
        manager.getEntityGraph("publisher.books");
        final String jpql =
            "select p from Publisher p where name = 'Addison'";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        query.setHint("javax.persistence.fetchgraph", graph);
        final Publisher p = query.getSingleResult();
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}
```

Wir können Entity-Graphen auch bei der `EntityManagerFactory` unter einem bestimmten Namen registrieren – via `addNamedEntityGraph`. Über den `EntityManager` kann dann mittels der Angabe dieses Namens der registrierte Graph ermittelt werden – via `getEntityGraph`. (Man beachte: der Graph wird bei der Factory registriert – abgeholt aber wird er vom Manager).

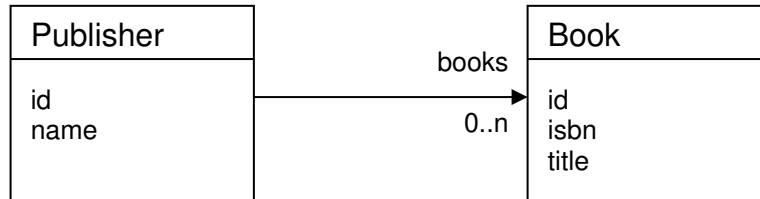
Die Ausgaben:

```
Hibernate: select ... from Publisher ... left outer join Book ...
Publisher [1, Addison]
    Book [1, 1111, Pascal]
    Book [3, 3333, Oberon]
    Book [2, 2222, Modula]
```

Die Service-Schicht einer Anwendung könnte nun ein- und dieselbe Methode der DAO-Schicht aufrufen und diese mit verschiedenen Entity-Graph-Namen parametrisieren. Abhängig davon würden der Service-Schicht dann verschiedene Objekt-Wolken (mit oder ohne `Books` ...) zugestellt werden können.

13.2 Named graphs

Die Entity-Klassen implementieren dieselbe Beziehung wie im letzten Abschnitt:



Auch das Datenbankschema ist dasselbe wie im letzten Abschnitt.

Publisher

Wir können Entity-Graphen auch mittels Annotationen erstellen. Eine `@Entity`-Klasse kann mit `@NamedEntityGraphs` (Plural) gekennzeichnet sein – einer Annotation, die ihrerseits `@NamedEntityGraph`-Annotationen (Singular) enthält. Wir definieren im folgenden zwei Graphen: einen Graphen ohne Knoten und einen zweiten mit dem "books"-Knoten. Für die Namen dieser Graphen benutzen wir Konstanten, die in der `Publisher`-Klasse definiert sind (`PUBLISHER_ONLY` resp. `PUBLISHER_WITH_BOOKS`):

```

package domain;
// ...
import javax.persistence.NamedAttributeNode;
import javax.persistence.NamedEntityGraph;
import javax.persistence.NamedEntityGraphs;

@Entity

@NamedEntityGraphs({
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_ONLY,
        attributeNodes = {
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_BOOKS,
        attributeNodes = {
            @NamedAttributeNode("books")
        }
    )
})

public class Publisher {

    public static final String PUBLISHER_ONLY =
        "PUBLISHER_ONLY";
    public static final String PUBLISHER_WITH_BOOKS =
        "PUBLISHER_WITH_BOOKS";
  
```

```
// ...

@OneToMany(fetch=FetchType.LAZY)
@JoinColumn(name="PUBLISHER_ID") // required!!!
private Set<Book> books = new HashSet<>();

// ...
}
```

Application

```
static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", p1));
        manager.persist(new Book("2222", "Modula", p1));
        manager.persist(new Book("3333", "Oberon", p1));
        manager.persist(new Book("4444", "Eiffel", p2));
    });
}
```

Die folgende `query`-Methode ist mit einer `graphId` parametrisiert. Sie erzeugt ein `TypedQuery`-Objekt, welches zunächst die "logische" Struktur der Abfrage enthält. Dann wird dieses `TypedQuery`-Objekt via `setHint` mit dem durch `graphId` bezeichneten Entity-Graphen konfiguriert:

```
public static Publisher query(
    EntityManager manager, String graphId) {
    final EntityGraph<?> graph = manager.getEntityGraph(graphId);
    final String jpql =
        "select p from Publisher p where name = 'Addison'";
    final TypedQuery<Publisher> query =
        manager.createQuery(jpql, Publisher.class);
    query.setHint("javax.persistence.fetchgraph", graph);
    return query.getSingleResult();
}
```

Die folgende `demo`-Methode ruft `query` mit `PUBLISHER_ONLY` auf (wir können dann natürlich auch nur den zurückgelieferten `Publisher` ausgeben):

```
static void demoQueryPublisherOnly(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p =
            query(manager, Publisher.PUBLISHER_ONLY);
        System.out.println(p);
    });
}
```

```
Hibernate: select ... from Publisher  
Publisher [1, Addison]
```

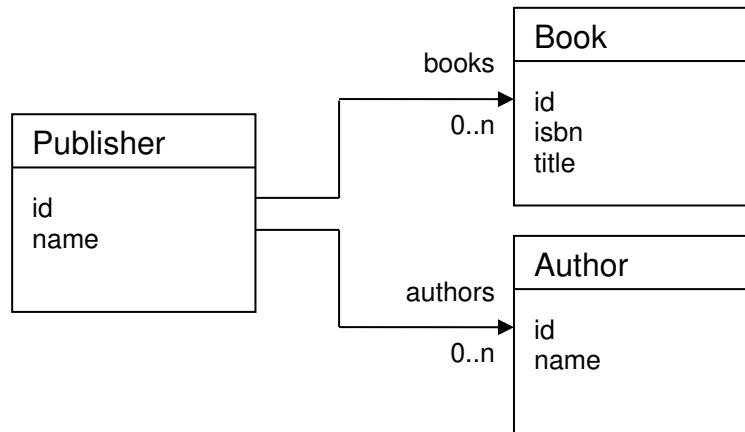
Die folgende Methode ruft `query` mit `PUBLISHER_WITH_BOOKS` auf – wir können anschließend den `Publisher` und seine `Books` ausgeben (ohne dass lazy loading stattfindet):

```
static void demoQueryPublisherWithBooks(TransactionTemplate tt) {  
    tt.run(manager -> {  
        final Publisher p =  
            query(manager, Publisher.PUBLISHER_WITH_BOOKS);  
        System.out.println(p);  
        p.getBooks().forEach(b -> System.out.println("\t" + b));  
    });  
}
```

```
Hibernate: select ... from Publisher left outer join Book ...  
Publisher [1, Addison]  
    Book [3, 3333, Oberon]  
    Book [1, 1111, Pascal]  
    Book [2, 2222, Modula]
```

13.3 Mehrere Attribute

Ein Publisher habe N Books und N AuthorS:



create.sql

```

create table PUBLISHER (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PUBLISHER_ID integer,
    primary key (ID),
    unique (ISBN),
    foreign key (PUBLISHER_ID) references PUBLISHER
);

create table AUTHOR (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    PUBLISHER_ID integer,
    primary key (ID),
    unique (NAME),
    foreign key (PUBLISHER_ID) references PUBLISHER
);
  
```

Author

```

package domain;
// ...
  
```

```
@Entity
public class Author {

    // Attribute: id, name ...

    // Konstruktoren, setter, getter, toString ...
}
```

Book

```
package domain;
// ...
@Entity
public class Book {

    // Attribute: id, isbn, title ...

    // Konstruktoren, setter, getter, toString ...
}
```

Publisher

```
package domain;
// ...

@Entity

@NamedEntityGraphs({
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_ONLY,
        attributeNodes = {
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_BOOKS,
        attributeNodes = {
            @NamedAttributeNode("books")
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_AUTHORS,
        attributeNodes = {
            @NamedAttributeNode("authors")
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_BOOKS_AND_AUTHORS,
        attributeNodes = {
            @NamedAttributeNode("books"),
            @NamedAttributeNode("authors")
        }
    )
})
```

```

public class Publisher {

    public static final String PUBLISHER_ONLY =
        "PUBLISHER_ONLY";
    public static final String PUBLISHER_WITH_BOOKS =
        "PUBLISHER_WITH_BOOKS";
    public static final String PUBLISHER_WITH_AUTHORS =
        "PUBLISHER_WITH_AUTHORS";
    public static final String PUBLISHER_WITH_BOOKS_AND_AUTHORS =
        "PUBLISHER_BOOKS_AND_AUTHORS";

    // ...

    @OneToMany(fetch = FetchType.LAZY)
    @JoinColumn(name = "PUBLISHER_ID") // required!!!
    private Set<Book> books = new HashSet<Book>();

    @OneToMany(fetch = FetchType.LAZY)
    @JoinColumn(name = "PUBLISHER_ID") // required!!!
    private Set<Author> authors = new HashSet<Author>();

    // ...
}

```

Der letzte `@NamedEntityGraph (PUBLISHER_WITH_BOOKS_AND_AUTHORS)` hat zwei Knoten: einen für das `books`-Attribut und einen zweiten für das `authors`-Attribut des `Publisher`s. Wenn nun ein Query mit diesem Graphen ausgeführt wird, werden zusammen mit dem `Publisher` auch sofort dessen `Books` und `Authors` geladen.

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", p1));
        manager.persist(new Book("2222", "Modula", p1));
        manager.persist(new Book("3333", "Oberon", p1));
        manager.persist(new Book("4444", "Eiffel", p2));
        manager.persist(new Author("Wirth", p1));
        manager.persist(new Author("Meyer", p1));
    });
}

```

Hier zunächst wieder die mit einer `graphId` parametrisierte `query`-Methode (es handelt sich um dieselbe Methode, die auch im letzten Abschnitt verwendet wurde):

```

public static Publisher query(
    EntityManager manager, String graphId) {

    final EntityGraph<?> graph = manager.getEntityGraph(graphId);
}

```



```

    final String jpql =
        "select p from Publisher p where name = 'Addison'";
    final TypedQuery<Publisher> query =
        manager.createQuery(jpql, Publisher.class);
    query.setHint("javax.persistence.fetchgraph", graph);
    return query.getSingleResult();
}

```

Hier einige demo-Methoden, in denen jeweils unterschiedliche Objekt-Wolken geladen werden:

```

static void demoQueryPublisherOnly(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p =
            query(manager, Publisher.PUBLISHER_ONLY);
        System.out.println(p);
    });
}

```

Die Ausgaben:

Hibernate: select ...
Publisher [1, Addison]

```

static void demoQueryPublisherWithBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p =
            query(manager, Publisher.PUBLISHER_WITH_BOOKS);
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}

```

Die Ausgaben:

Hibernate: select ...
Publisher [1, Addison]
 Book [1, 1111, Pascal]
 Book [2, 2222, Modula]
 Book [3, 3333, Oberon]

```

static void demoQueryPublisherWithAuthors(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p =
            query(manager, Publisher.PUBLISHER_WITH_AUTHORS);
        System.out.println(p);
        p.getAuthors().forEach(a -> System.out.println("\t" + a));
    });
}

```

Die Ausgaben:

```
Hibernate: select ...  
Publisher [1, Addison]  
    Author [2, Meyer]  
    Author [1, Wirth]
```

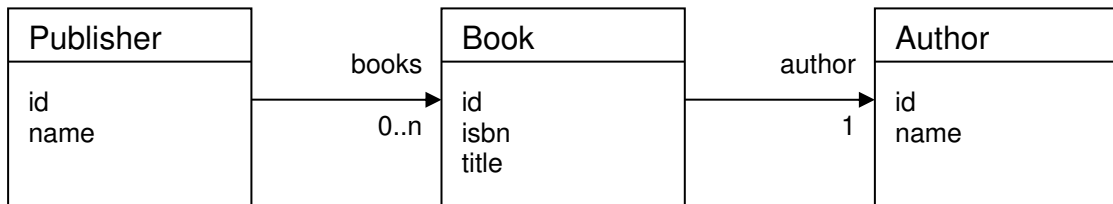
```
static void demoQueryPublisherWithBooksAndAuthors (  
    TransactionTemplate tt) {  
    tt.run(manager -> {  
        final Publisher p =  
            query(manager, Publisher.PUBLISHER_WITH_BOOKS_AND_AUTHORS);  
        System.out.println(p);  
        p.getBooks().forEach(b -> System.out.println("\t" + b));  
        p.getAuthors().forEach(a -> System.out.println("\t" + a));  
    });  
}
```

Die Ausgaben:

```
Hibernate: select ...  
Publisher [1, Addison]  
    Book [2, 2222, Modula]  
    Book [1, 1111, Pascal]  
    Book [3, 3333, Oberon]  
    Author [2, Meyer]  
    Author [1, Wirth]
```

13.4 Subgraphs

Ein Publisher hat n Books, jedes Book hat einen Author:



create.sql

```

create table PUBLISHER (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table AUTHOR (
    ID integer generated by default as identity (start with 1),
    NAME varchar (128) not null,
    primary key (ID),
    unique (NAME)
);

create table BOOK (
    ID integer generated by default as identity (start with 1),
    ISBN varchar (20) not null,
    TITLE varchar (128) not null,
    PUBLISHER_ID integer,
    AUTHOR_ID integer,
    primary key (ID),
    unique (ISBN),
    foreign key (PUBLISHER_ID) references PUBLISHER,
    foreign key (AUTHOR_ID) references AUTHOR
);
  
```

Author

```

package domain;
// ...
@Entity
public class Author {
    // Attribute id, name
    // Konstruktoren, getter, setter, toString ...
}
  
```

Book

```
package domain;
// ...
@Entity
public class Book {

    // Attribute id, isbn, title ...

    @ManyToOne(fetch = FetchType.LAZY)
    private Author author;

    Book() { }

    public Book(String isbn, String title,
        Publisher publisher, Author author) {
        this.isbn = isbn;
        this.title = title;
        publisher.getBooks().add(this);
        this.author = author;
    }

    // getter, setter, toString ...
}
```

Publisher

```
package domain;
// ...
@Entity
@NamedEntityGraphs({
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_ONLY,
        attributeNodes = {
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_BOOKS,
        attributeNodes = {
            @NamedAttributeNode("books")
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_BOOKS_WITH_AUTHORS,
        attributeNodes = {
            @NamedAttributeNode(value = "books", subgraph = "a")
        },
        subgraphs= {
            @NamedSubgraph(
                name = "a",
                attributeNodes = @NamedAttributeNode("author")
            )
        }
    )
},
),
```

```

}))

//@formatter:on

public class Publisher {

    public static final String PUBLISHER_ONLY =
        "PUBLISHER_ONLY";
    public static final String PUBLISHER_WITH_BOOKS =
        "PUBLISHER_WITH_BOOKS";
    public static final String PUBLISHER_WITH_BOOKS_WITH_AUTHORS =
        "PUBLISHER_WITH_BOOK_WITH_AUTHORS";

    // Attribute: id, name...

    @OneToMany(fetch = FetchType.LAZY)
    @JoinColumn(name = "PUBLISHER_ID") // required!!!
    private Set<Book> books = new HashSet<Book>();

    // Konstruktoren, getter, setter, toString ...
}

```

Der letzte Entity-Graph ist der interessanteste:

```

@NamedEntityGraph(
    name = Publisher.PUBLISHER_WITH_BOOKS_WITH_AUTHORS,
    attributeNodes = {
        @NamedAttributeNode(value = "books", subgraph = "a")
    },
    subgraphs= {
        @NamedSubgraph(
            name = "a",
            attributeNodes = @NamedAttributeNode("author")
        )
    }
),

```

Der Graph hat einen Knoten für die `books` des `Publisher`s. Diesem Knoten sollen nun weitere Knoten zugeordnet werden. Also hat der Knoten ein `subgraph`-Attribut, welches dem Subgraph einen Namen zuweist.

Unter `@subgraphs` wird dann ein `@NamedSubgraph` formuliert, welcher über das `name`-Attribut den `books`-Knoten referenziert. Diesem wird dann ein weiterer Knoten zugewiesen: der `author`-Knoten von `Book`.

Somit beschreibt der komplette Graph den Weg, der vom `Publisher` zu seinen `Books` und von diesen `Books` zu deren jeweiligem `Author` führt.

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {

```

```

        final Publisher p1 = new Publisher("Addison");
        final Publisher p2 = new Publisher("Prentice");
        final Author a1 = new Author("Wirth");
        final Author a2 = new Author("Meyer");
        manager.persist(a1);
        manager.persist(a2);
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", p1, a1));
        manager.persist(new Book("2222", "Modula", p1, a1));
        manager.persist(new Book("3333", "Oberon", p1, a2));
        manager.persist(new Book("4444", "Eiffel", p2, a2));
    });
}

```

Auch hier wird wieder die bereits aus den letzten Abschnitten bekannte `query`-Methode verwendet:

```

public static Publisher query(EntityManager manager, String graphId) {
    final EntityGraph<?> graph = manager.getEntityGraph(graphId);
    final String jpql =
        "select p from Publisher p where name = 'Addison'";
    final TypedQuery<Publisher> query =
        manager.createQuery(jpql, Publisher.class);
    query.setHint("javax.persistence.fetchgraph", graph);
    final Publisher p = query.getSingleResult();
    return p;
}

```

Hier einige `demo`-Methoden:

```

static void demoQueryPublisherOnly(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p =
            query(manager, Publisher.PUBLISHER_ONLY);
        System.out.println(p);
    });
}

```

Die Ausgaben:

Hibernate: select ...
Publisher [1, Addison]

```

static void demoQueryPublisherWithBooks(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p =
            query(manager, Publisher.PUBLISHER_WITH_BOOKS);
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}

```

Die Ausgaben:

```
Hibernate: select ... from Publisher left outer join Book ...
Publisher [1, Addison]
    Book [2, 2222, Modula]
    Book [1, 1111, Pascal]
    Book [3, 3333, Oberon]
```

```
static void demoQueryPublisherWithBooksAndAuthors(
    TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p =
            query(manager, Publisher.PUBLISHER_WITH_BOOKS_WITH_AUTHORS);
        System.out.println(p);
        p.getBooks().forEach(b -> {
            System.out.println("\t" + b);
            System.out.println("\t\t" + b.getAuthor());
        });
    });
}
```

Die Ausgaben:

```
Hibernate: select ... from Publisher
    left outer join Book left outer join Author ...
Publisher [1, Addison]
    Book [1, 1111, Pascal]
        Author [1, Wirth]
    Book [3, 3333, Oberon]
        Author [2, Meyer]
    Book [2, 2222, Modula]
        Author [1, Wirth]
```

Wir können den letzten Graphen natürlich auch programmatisch erzeugen:

```
static void demoQuerySubgraphExplicitCreation(
    TransactionTemplate tt) {
    tt.run(manager -> {

        final EntityGraph<Publisher> graph =
            manager.createEntityGraph(Publisher.class);
        Subgraph<?> sub = graph.addSubgraph("books");
        sub.addAttributeNodes("author");

        final String jpql =
            "select p from Publisher p where name = 'Addison'";
        final TypedQuery<Publisher> query =
            manager.createQuery(jpql, Publisher.class);
        query.setHint("javax.persistence.fetchgraph", graph);
        final Publisher p = query.getSingleResult();
        System.out.println(p);
        p.getBooks().forEach(b -> {
            System.out.println("\t" + b);
            System.out.println("\t\t" + b.getAuthor());
        });
    });
}
```

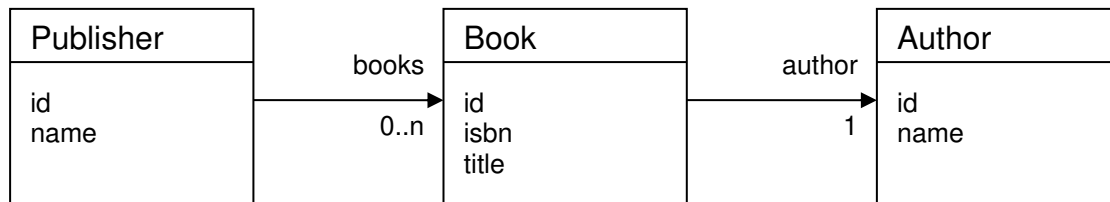
```
        });  
    });  
}
```

Die Ausgaben sind dieselben wie bei der letzten `demo-Methode`.

13.5 Verteilte Graphen

Wir können die Entity-Graphen auf verschiedene Entity-Klassen verteilen.

Auch im folgenden Beispiel hat ein `Publisher` `n` `Books` und jedes `Book` hat einen `Author`:



Das Datenbank-Schema ist dasselbe wie im letzten Abschnitt. Auch die Klassen sind dieselben wie die im letzten Abschnitt – mit Ausnahme der Annotationen.

Author

```

package domain;
// ...
@Entity
public class Author {
    // Attribute id, name
    // Konstruktoren, getter, setter, toString ...
}
  
```

Book

```

package domain;
// ...

@NamedEntityGraphs({
    @NamedEntityGraph(name = Book.BOOK_ONLY, attributeNodes = {
    }),
    @NamedEntityGraph(name = Book.BOOK_WITH_AUTHOR, attributeNodes = {
        @NamedAttributeNode("author")
    })
})

@Entity
public class Book {

    public static final String BOOK_ONLY = "BOOK_ONLY";
    public static final String BOOK_WITH_AUTHOR = "BOOK_WITH_AUTHOR";

    // Attribute id, isbn, title...
}
  
```

```

    @ManyToOne(fetch = FetchType.LAZY)
    private Author author;

    Book() { }

    public Book(String isbn, String title,
        Publisher publisher, Author author) {
        this.isbn = isbn;
        this.title = title;
        publisher.getBooks().add(this);
        this.author = author;
    }

    // getter, setter, toString...
}

```

Publisher

```

package domain;
// ...
@Entity
@NamedEntityGraphs({
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_ONLY,
        attributeNodes = {
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_BOOKS,
        attributeNodes = {
            @NamedAttributeNode("books")
        }
    ),
    @NamedEntityGraph(
        name = Publisher.PUBLISHER_WITH_BOOKS_WITH_AUTHORS,
        attributeNodes = {
            @NamedAttributeNode(
                value = "books",
                subgraph = Book.BOOK_WITH_AUTHOR
            )
        }
    ),
})
public class Publisher {

    public static final String PUBLISHER_ONLY =
        "PUBLISHER_ONLY";
    public static final String PUBLISHER_WITH_BOOKS =
        "PUBLISHER_WITH_BOOKS";
    public static final String PUBLISHER_WITH_BOOKS_WITH_AUTHORS =
        "PUBLISHER_WITH_BOOK_WITH_AUTHORS";

    // Attribute: id, name...
}

```

```

@OneToMany(fetch = FetchType.LAZY)
@JoinColumn(name = "PUBLISHER_ID") // required!!!
private Set<Book> books = new HashSet<Book>();

// Konstruktoren, getter, setter, toString ...
}

```

Wir beziehen uns jetzt in den Entity-Graph-Definitionen der Klasse Publisher auf diejenigen der Klasse Book.

Wird nun aber folgende demo-Methode aufgeführt:

```

static void demoQueryPublisherWithBooksAndAuthors(
    TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p = query(
            manager, Publisher.PUBLISHER_WITH_BOOKS_WITH_AUTHORS);
        System.out.println(p);
        p.getBooks().forEach(b -> {
            System.out.println("\t" + b);
            System.out.println("\t\t" + b.getAuthor());
        });
    });
}

```

so findet nun zweimal lazy loading statt:

```

Hibernate: select ...
Publisher [1, Addison]
    Book [1, 1111, Pascal]
Hibernate: select ...
    Author [1, Wirth]
    Book [3, 3333, Oberon]
Hibernate: select ...
    Author [2, Meyer]
    Book [2, 2222, Modula]
    Author [1, Wirth]

```

13.6 Loadgraph vs. fetchgraph

"If a fetch graph is used, only the attributes specified by the entity graph will be treated as `FetchType.EAGER`. All other attributes will be lazy. If a load graph is used, all attributes that are not specified by the entity graph will keep their default fetch type."

Der Unterschied beider Attribute liegt dabei in der Behandlung von Attributen, die nicht im jeweiligen Entity-Graphen angegeben sind: Während beim Fetch-Graphen alle nicht angegebenen Attribute automatisch lazy sind, behalten sie beim Load-Graphen ihr im Mapping konfiguriertes Fetch-Verhalten und werden, falls sie als `EAGER` gemappt sind, zusätzlich zu den im Load-Graphen angegebenen Attributen geladen.

Um das Verhalten genauer zu verstehen, sollte der Leser / die Leserin das entsprechende Projekt des Eclipse-Workspaces ausprobieren – er / sie wird allerdings enttäuscht sein: das Verhalten bei "fetchgraph" ist dasselbe wie das Verhalten bei "loadgraph". Das Projekt wird daher auch in diesem Skript nicht näher vorgestellt...

13.7 Ein Graph-Konzept

Sowohl die "manuelle" Konstruktion als auch die Annotation-basierte Konstruktion eines `EntityGraph`-Objekts ist recht mühsam. Im folgenden wird ein recht einfach Konzept benutzt, mittels dessen `EntityGraph`-Objekte automatisch erzeugt werden können.

Das Konzept beruht auf einer einfachen Überlegung: Wir registrieren `EntityGraph`-Objekte unter einer ID, aus der die Struktur des Entity-Graphen hervorgeht: aus der ID kann automatisch der `EntityGraph` berechnet werden, der für diese ID registriert werden soll.

Sei z.B. folgende ID gegeben: `books(publisher(city), content)`

Dann kann daraus das folgende `EntityGraph`-Objekt (mitsamt seiner `Subgraph`-Objekte) berechnet werden:

```
books
  publisher
    city
  content
```

Wir benutzen eine kleine Hilfsklasse `Graph`, von welcher hier nur die öffentliche Schnittstelle dargestellt wird:

```
package util.graph;

public class Graph<T> implements Serializable {

    public final Class<T> rootType;
    public final String name;

    public Graph(Class<T> rootType, String name) { ... }
    public Graph(Class<T> rootType) { ... }

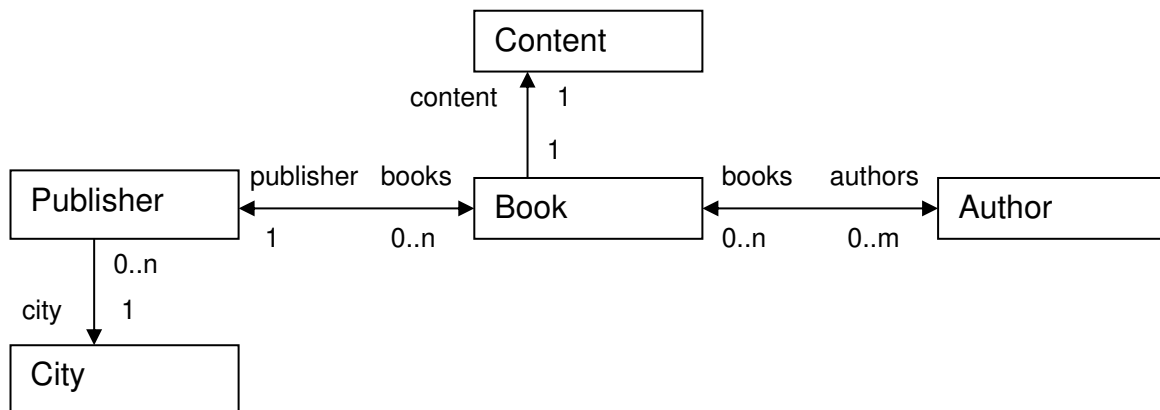
    public void register(EntityManager manager) { ... }
}
```

Der erste Parameter, der an den Konstruktor übergeben wird, ist die Klasse des Wurzelobjekts. Und der an den Konstruktor übergebene ID (`name`) muss folgenden Aufbau gehorchen:

```
Name ::= [ NodeList ]
NodeList ::= [ Node { "," Node } ]
Node ::= identifier [ "(" NodeList ")" ]
```

In der `register`-Methode wird aufgrund eines solchen Namens der `EntityGraph` berechnet und bei einem `EntityManager` registriert.

Wir demonstrieren die Verwendung des `Graph`-Konzepts anhand der folgenden Entity-Klassen:



Wir definieren ausgehend vom `Publisher` einen leeren `Graph`:

```
public static final Graph<Publisher> publisher =
    new Graph<Publisher>(Publisher.class);
```

Wir definieren ausgehend vom `Publisher` einen `Graph`, der die `books` des `Publishers` umspannt:

```
public static final Graph<Publisher> publisherBooks =
    new Graph<Publisher>(Publisher.class,
        "books");
```

Der folgende `Graph` geht vom `Publisher` aus und enthält die `books` des `Publishers` – und zu jedem `Book` die `authors`:

```
public static final Graph<Publisher> publisherBooksAuthors =
    new Graph<Publisher>(Publisher.class,
        "books(authors)");
```

Der folgende `Graph` hat die Wurzelklasse `Book`. Der `Graph` umfasst den `publisher` des `Books`, die `city` des `Publishers` und den `content` des `Books`:

```
public static final Graph<Book> bookPublisherCityAuthorContent =
    new Graph<Book>(Book.class,
        "publisher(city), authors, content");
```

Und schließlich definieren wird einen Graph, der von `Author` ausgeht. Er umfasst die `books` des `Authors` und zu jedem `Book` den `publisher` (und die `city` des `Publishers`) und den `content`:

```
public static final Graph<Author> authorBooksPublisherCityContent =
    new Graph<Author>(Author.class,
        "books(publisher(city), content)");
```

Alle in einem Namen enthaltenen Begriffe (`books`, `publisher` etc.) sind die Attribut-Namen der entsprechenden Klassen (resp. die Property-Namen).

Aus den obigen Graph-Objekten erzeugen wird nun die `EntityGraph`-Objekte (mitsamt der erforderlichen Subgraphs) und registrieren diese bei einem `EntityManager`:

```
static void createGraphs(TransactionTemplate tt) {
    tt.run(manager -> {

        publisher.register(manager);
        publisherBooks.register(manager);
        publisherBooksAuthors.register(manager);
        bookPublisherCityAuthorContent.register(manager);
        authorBooksPublisherCityContent.register(manager);

        EntityGraphUtil.printGraphs(manager);
    });
}
```

Hier die Ausgaben der `printGraphs`-Methode:

```
| domain.Publisher {
|   "" {
|   }
|   "books(authors)" {
|       Publisher.books [Set<Book>]
|       Book.authors [Set<Author>]
|   }
|   "books" {
|       Publisher.books [Set<Book>]
|   }
| }
| domain.Author {
|   "books(publisher(city),content)" {
|       Author.books [Set<Book>]
|       Book.publisher [Publisher]
|       Publisher.city [City]
|       Book.content [Content]
|   }
| }
| domain.Content {
| }
```

```
| domain.Book {
|     "publisher(city),authors,content" {
|         Book.publisher [Publisher]
|         Publisher.city [City]
|         Book.content [Content]
|         Book.authors [Set<Author>]
|     }
| }
| domain.City {
| }
```

Wir benutzen für die folgenden demo-Methoden eine allgemeine query-Methode, der neben dem EntityManager jeweils ein Graph-Objekt übergeben wird – eine query-Methode, die einer Transaktion erzeugt und in dieser Transaktion die geforderte Objekt-Wolke erzeugt:

```
public static <T> List<T> query(
    TransactionTemplate tt, Graph<T> graphId) {
    return tt.runWithResult(manager -> {
        final EntityGraph<?> graph = manager.getEntityGraph(graphId.name);
        final String jpql = "select distinct a from " +
            graphId.rootType.getSimpleName() + " a";
        final TypedQuery<T> query = manager.createQuery(jpql,
            graphId.rootType);
        query.setHint("javax.persistence.fetchgraph", graph);
        return query.getResultList();
    });
}
```

Die Datenbank wird bereits über das create.sql-Sript gefüllt – dort werden bereits die SQL-INSERTs ausgeführt.

Hier schließlich die demo-Methoden und ihre Ausgaben (die Ausgaben erfolgen außerhalb der jeweiligen Transaktion – es findet also kein lazy loading statt):

```
static void demo1(TransactionTemplate tt) {
    final List<Publisher> publishers =
        query(tt, publisherBooks);
    publishers.forEach(p -> System.out.println(p));
}
```

```
Publisher [1, Addison]
Publisher [2, Prentice]
```

```
static void demo2(TransactionTemplate tt) {
    final List<Publisher> publishers =
        query(tt, publisherBooks);
    publishers.forEach(p -> {
        System.out.println(p);
        p.getBooks().forEach(b -> System.out.println("\t" + b));
    });
}
```



```
}
```

```
Publisher [1, Addison]
    Book [1, 1111, 10.0, Pascal]
    Book [2, 2222, 20.0, Modula]
Publisher [2, Prentice]
    Book [3, 3333, 30.0, Oberon]
```

```
static void demo3(TransactionTemplate tt) {
    final List<Publisher> publishers =
        query(tt, publisherBooksAuthors);
    publishers.forEach(p -> {
        System.out.println(p);
        p.getBooks().forEach(b -> {
            System.out.println("\t" + b);
            b.getAuthors().forEach(
                a -> System.out.println("\t\t" + a));
        });
    });
}
```

```
Publisher [1, Addison]
    Book [1, 1111, 10.0, Pascal]
        Author [1, Wirth]
    Book [2, 2222, 20.0, Modula]
        Author [1, Wirth]
        Author [2, Meyer]
Publisher [2, Prentice]
    Book [3, 3333, 30.0, Oberon]
        Author [1, Wirth]
        Author [2, Meyer]
```

```
static void demo4(TransactionTemplate tt) {
    final List<Book> books =
        query(tt, bookPublisherCityAuthorContent);
    books.forEach(b -> {
        System.out.println(b);
        System.out.println("\t" + b.getPublisher());
        System.out.println("\t\t" + b.getPublisher().getCity());
        System.out.println("\t" + b.getContent());
        b.getAuthors().forEach(a -> System.out.println("\t" + a));
    });
}
```

```
Book [1, 1111, 10.0, Pascal]
    Publisher [1, Addison]
        City [1, Berlin]
    Content [1, Programmiersprache Pascal]
    Author [1, Wirth]
Book [2, 2222, 20.0, Modula]
    Publisher [1, Addison]
        City [1, Berlin]
    Content [2, Programmiersprache Modula]
```

```

    Author [2, Meyer]
    Author [1, Wirth]
Book [3, 3333, 30.0, Oberon]
    Publisher [2, Prentice]
        City [2, New York]
    Content [3, Programmiersprache Oberon]
    Author [2, Meyer]
    Author [1, Wirth]

```

```

static void demo5(TransactionTemplate tt) {
    final List<Author> authors =
        query(tt, authorBooksPublisherCityContent);
    authors.forEach(a -> {
        System.out.println(a);
        a.getBooks().forEach(b -> {
            System.out.println("\t" + b);
            System.out.println("\t\t" + b.getPublisher());
            System.out.println("\t\t\t" +
                b.getPublisher().getCity());
            System.out.println("\t\t" + b.getContent());
        });
    });
}

```

```

Author [1, Wirth]
    Book [3, 3333, 30.0, Oberon]
        Publisher [2, Prentice]
            City [2, New York]
        Content [3, Programmiersprache Oberon]
    Book [1, 1111, 10.0, Pascal]
        Publisher [1, Addison]
            City [1, Berlin]
        Content [1, Programmiersprache Pascal]
    Book [2, 2222, 20.0, Modula]
        Publisher [1, Addison]
            City [1, Berlin]
        Content [2, Programmiersprache Modula]
Author [2, Meyer]
    Book [3, 3333, 30.0, Oberon]
        Publisher [2, Prentice]
            City [2, New York]
        Content [3, Programmiersprache Oberon]
    Book [2, 2222, 20.0, Modula]
        Publisher [1, Addison]
            City [1, Berlin]
        Content [2, Programmiersprache Modula]

```

14 **SqlResultSetMapping**

JPA 2.1 führt einige weitere Möglichkeiten ein, mittels derer die Resultate einer nativen (SQL-) Abfrage auf Pojo-Objekte abgebildet werden kann.

14.1 Constructor Expressions

Sei folgende Datenbank gegeben:

create.sql

```
create table BOOK (  
    F_ID integer generated by default as identity (start with 1),  
    F_ISBN varchar (20) not null,  
    F_TITLE varchar (128) not null,  
    F_PRICE double not null,  
    primary key (F_ID),  
    unique (F_ISBN)  
)
```

Wir definieren zwei Nicht-@Entity-Klassen (einfache "Pojo"-Klassen), die für Constructor-Queries genutzt werden sollen:

BookIsbnPrice

```
package domain;  
  
public class BookIsbnPrice {  
  
    private String isbn;  
    private double price;  
  
    public BookIsbnPrice(String isbn, double price) { ... }  
  
    // getter, setter, toString...  
}
```

BookIsbnTitle

```
package domain;  
  
public class BookIsbnTitle {  
  
    private String isbn;  
    private String title;  
  
    public BookIsbnTitle(String isbn, String title) { ... }  
  
    // getter, setter, toString...  
}
```

Book

Die Klasse `Book` ist eine `@Entity`-Klasse, die in der folgenden Anwendung aber nur zum Zwecke der Persistierung von `Book`-Objekten genutzt wird.

Die Klasse ist allerdings zusätzlich mit einer `@SqlResultSetMappings`-Annotation ausgestattet. Diese Annotation enthält zwei `@SqlResultSetMapping`-Annotationen (Singulär).

Jedes `@SqlResultSetMapping` enthält einen Namen, die Klasse der zu erzeugenden Pojos und die Namen derjenigen Spalten, deren Werte dem jeweiligen Pojo-Konstruktor übergeben werden sollen:

```
package domain;
// ...
import javax.persistence.ConstructorResult;
import javax.persistence.SqlResultSetMapping;
import javax.persistence.SqlResultSetMappings;

@Entity

@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "BookIsbnTitleMapping",
        classes = @ConstructorResult(
            targetClass = BookIsbnTitle.class,
            columns = {
                @ColumnResult(name = "f_isbn"),
                @ColumnResult(name = "f_title"),
            }
        )
    ),
    @SqlResultSetMapping(
        name = "BookIsbnPriceMapping",
        classes = @ConstructorResult(
            targetClass = BookIsbnPrice.class,
            columns = {
                @ColumnResult(name = "f_isbn", type = String.class),
                @ColumnResult(name = "f_price", type = double.class)
            }
        )
    )
})

public class Book {

    @Id
    @Column(name = "f_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    @Column(name = "f_isbn")
    private String isbn;
```

```

@Basic
@Column(name = "f_title")
private String title;

@Basic
@Column(name = "f_price")
private double price;

// Konstruktoren, getter, setter, toString...
}

```

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        manager.persist(new Book("1111", "Pascal", 10.0));
        manager.persist(new Book("2222", "Modula", 20.0));
    });
}

```

Die beiden folgenden Methoden rufen jeweils `createNativeQuery` auf – übergeben dabei aber jeweils den Namen eines `@SqlResultSetMappings`:

In der ersten `demo`-Methode werden `BookIsbnTitle`-Objekte erzeugt;

```

static void demoQueryBookIsbnTitle(TransactionTemplate tt) {
    tt.run(manager -> {
        final String sql = "select b.f_isbn, b.f_title from Book b";
        final Query query =
            manager.createNativeQuery(sql, "BookIsbnTitleMapping");
        final List<BookIsbnTitle> bookValues = query.getResultList();
        bookValues.forEach(System.out::println);
    });
}

```

Die Ausgaben:

```

BookIsbnTitle [1111, Pascal]
BookIsbnTitle [2222, Modula]

```

In der zweiten `demo`-Methode werden `BookIsbnPrice`-Objekte erzeugt;

```

static void demoQueryBookIsbnPrice(TransactionTemplate tt) {
    tt.run(manager -> {
        final String sql = "select b.f_isbn, b.f_price from Book b";
        final Query query =
            manager.createNativeQuery(sql, "BookIsbnPriceMapping");
        final List<BookIsbnPrice> bookValues = query.getResultList();
        bookValues.forEach(System.out::println);
    });
}

```

Die Ausgaben:

```
BookIsbnPrice [1111, 10.0]  
BookIsbnPrice [2222, 20.0]
```

14.2 Constructor Expressions - Associations

Ein Book habe nun eine Referenz auf einen Publisher:



Wir wollen Pojo-Objekt erzeugen, von denen jedes die ISBN und den Titel des Buches enthält und den Namen des Verlages.

create.sql

```

create table PUBLISHER (
    F_ID integer generated by default as identity (start with 1),
    F_NAME varchar (20) not null,
    F_CITY varchar (20) not null,
    primary key (F_ID),
)
  
```

```

create table BOOK (
    F_ID integer generated by default as identity (start with 1),
    F_ISBN varchar (20) not null,
    F_TITLE varchar (128) not null,
    F_PRICE double not null,
    F_PUBLISHER_ID integer not null,
    primary key (F_ID),
    foreign key (F_PUBLISHER_ID) references PUBLISHER,
    unique (F_ISBN)
)
  
```

BookIsbnTitlePublisherName

Wir verwenden nun eine Pojo-Klasse, zu deren Instanziierung dem Konstruktor sowohl Werte von BOOK-Spalten als auch ein Wert einer PUBLISHER-Spalte übergeben werden müssen:

```

package domain;

public class BookIsbnTitlePublisherName {

    private String isbn;
    private String title;
    private String publisherName;

    // Konstruktor, getter, setter, toString ...
  
```



```
}
```

Publisher

Die Klasse `Publisher` ist trivial:

```
package domain;

@Entity
public class Publisher {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "F_ID")
    private Integer id;

    @Basic
    @Column(name = "F_NAME")
    private String name;

    @Basic
    @Column(name = "F_CITY")
    private String city;

    // Konstruktoren, getter, setter, toString ...
}
```

Book

In der `Book`-Klasse wird ein `@SqlResultSetMapping` definiert, welches dazu benutzt werden wird, drei Spaltenwerte eines Result-Sets auf eine neue Instanz der Klasse `BookIsbnTitlePublisherName` abzubilden:

```
package domain;
// ...

@Entity

@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "BookIsbnTitlePublisherNameMapping",
        classes = @ConstructorResult(
            targetClass = BookIsbnTitlePublisherName.class,
            columns = {
                @ColumnResult(name = "f_isbn"),
                @ColumnResult(name = "f_title"),
                @ColumnResult(name = "f_name"),
            }
        )
    )
})

public class Book {
```

```

@Id
@Column(name = "f_id")
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;

@Basic
@Column(name = "f_isbn")
private String isbn;

@Basic
@Column(name = "f_title")
private String title;

@Basic
@Column(name = "f_price")
private double price;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name="f_publisher_id")
private Publisher publisher;

// Konstruktoren, getter, setter, toString ...
}

```

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        final Publisher p1 = new Publisher("Addison", "Berlin");
        final Publisher p2 = new Publisher("Prentice", "New York");
        manager.persist(p1);
        manager.persist(p2);
        manager.persist(new Book("1111", "Pascal", 30, p1));
        manager.persist(new Book("2222", "Modula", 40, p1));
        manager.persist(new Book("3333", "Oberon", 50, p1));
        manager.persist(new Book("4444", "Eiffel", 20, p2));
    });
}

```

Der folgende SQL-Select selektiert drei Spalten:

```

f_isbn (von b)
f_title (von b)
f_name (von p)

```

Die Werte dieser drei Spalten werden dann dem Konstruktor der Pojo-Klasse übergeben:

```

static void demoQueryBookIsbnTitlePublisherName(
    TransactionTemplate tt) {
    tt.run(manager -> {
        final String sql = "select " +

```

```
        "b.f_isbn, b.f_title, p.f_name from Book b " +  
        "join Publisher p ON b.f_publisher_id = p.f_id";  
        final Query query = manager.createNativeQuery(  
            sql, "BookIsbnTitlePublisherNameMapping");  
        final List<BookIsbnTitlePublisherName> bookValues =  
            query.getResultList();  
        bookValues.forEach(System.out::println);  
    });  
}
```

Die Ausgaben:

```
BookIsbnTitlePublisherName [1111, Pascal, Addison]  
BookIsbnTitlePublisherName [2222, Modula, Addison]  
BookIsbnTitlePublisherName [3333, Oberon, Addison]  
BookIsbnTitlePublisherName [4444, Eiffel, Prentice]
```

14.3 EntityResult

Das letzte Beispiel sei dem Leser / der Leserin zum Selbststudium überlassen...

Book
id isbn title authorId

Author
id name

create.sql

```
create table AUTHOR (
  F_ID integer generated by default as identity (start with 1),
  F_NAME varchar (128) not null,
  primary key (F_ID),
  unique (F_NAME)
)
```

```
create table BOOK (
  F_ID integer generated by default as identity (start with 1),
  F_ISBN varchar (20) not null,
  F_TITLE varchar (128) not null,
  F_PRICE double not null,
  F_AUTHOR_ID integer,
  primary key (F_ID),
  unique (F_ISBN),
  foreign key (F_AUTHOR_ID) references AUTHOR
)
```

Author

```
package domain;
// ...
@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "f_id")
    private Integer id;

    @Basic
    @Column(name = "f_name")
    private String name;

    // Konstruktoren, setter, getter, toString ...
}
```

Book

```

package domain;
// ...
import javax.persistence.EntityResult;
import javax.persistence.FieldResult;

@Entity

@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "BookMapping",
        entities = {
            @EntityResult(
                entityClass = Book.class,
                fields = {
                    @FieldResult(name = "id", column = "f_id"),
                    @FieldResult(name = "isbn", column = "f_isbn"),
                    @FieldResult(name = "title", column = "f_title"),
                    @FieldResult(name = "price", column = "f_price"),
                    @FieldResult(name = "authorId", column = "f_author_id")
                }
            ),
            @EntityResult(
                entityClass = Author.class,
                fields = {
                    @FieldResult(name = "id", column = "f_id"),
                    @FieldResult(name = "name", column = "f_name")
                }
            )
        }
    )
})

public class Book implements {

    @Id
    @Column(name = "f_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    @Column(name = "f_isbn")
    private String isbn;

    @Basic
    @Column(name = "f_title")
    private String title;

    @Basic
    @Column(name = "f_price")
    private double price;

    @Basic

```

```

@Column(name = "f_author_id")
private Integer authorId;

Book() { }

public Book(String isbn, String title, double price,
            Integer authorId) { ... }

// setter, getter, toString ...
}

```

Application

```

static void demoPersist(TransactionTemplate tt) {
    tt.run(manager -> {
        Author author = new Author("Wirth");
        manager.persist(author);
        manager.persist(new Book("1111", "Pascal", 10.0, author.getId()));
        manager.persist(new Book("2222", "Modula", 20.0, author.getId()));
    });
}

```

```

AUTHOR
F_ID F_NAME
-----
1    Wirth
-----

```

```

BOOK
F_ID F_ISBN F_TITLE F_PRICE F_AUTHOR_ID
-----
1    1111   Pascal   10.0    1
2    2222   Modula   20.0    1
-----

```

```

static void demoQuerySimple(TransactionTemplate tt) {
    tt.run(manager -> {
        final String sql = "select " +
            "b.f_id, b.f_isbn, b.f_title, b.f_price, b.f_author_id " +
            "from Book b left outer join Author a " +
            "on a.f_id = b.f_author_id";
        final Query query = manager.createNativeQuery(sql, Book.class);
        final List<Book> books = query.getResultList();
        books.forEach(b -> {
            System.out.println(b);
        });
        if (books.size() > 0)
            System.out.println(manager.contains(books.get(0)));
    });
}

```

```

Book [1, 1111, Pascal, 10.0, 1]
Book [2, 2222, Modula, 20.0, 1]

```

true

```
@SuppressWarnings("unchecked")
static void demoQueryWithAlias(TransactionTemplate tt) {
    tt.run(manager -> {
        final String sql = "select " +
            "b.f_id as id, b.f_isbn as isbn, b.f_title as title, " +
            "b.f_price as price, b.f_author_id as author " +
            "from Book b left outer join Author a " +
            "on a.f_id = b.f_author_id";
        final Query query = manager.createNativeQuery(sql, Book.class);
        final List<Book> books = query.getResultList();
        books.forEach(b -> {
            System.out.println(b);
        });
        if (books.size() > 0)
            System.out.println(manager.contains(books.get(0)));
    });
}
```

Book [1, 1111, Pascal, 10.0, 1]

Book [2, 2222, Modula, 20.0, 1]

true

```
@SuppressWarnings("unchecked")
static void demoQueryWithMapping(TransactionTemplate tt) {
    tt.run(manager -> {
        final String sql = "select " +
            "b.f_id, b.f_isbn, b.f_title, b.f_price, b.f_author_id, " +
            "a.f_id, a.f_name " +
            "from Book b left outer join Author a " +
            "on a.f_id = b.f_author_id";
        final Query query = manager.createNativeQuery(sql, "BookMapping");

        final List<Object[]> books = query.getResultList();
        books.forEach((Object[] objects) -> {
            System.out.println(objects[0] + " " +
                manager.contains(objects[0]));
            System.out.println(objects[1] + " " +
                manager.contains(objects[1]));
        });
    });
}
```

Book [1, 1111, Pascal, 10.0, 1] true

Author [1, Wirth] true

Book [2, 2222, Modula, 20.0, 1] true

Author [2, Wirth] true

15 Metadaten

JPA ermöglicht den Einblick in die Metadaten: wie sind die Entity-Klassen annotiert?

Im ersten Abschnitt wird gezeigt, wie die Metadaten ermittelt werden können.

Im zweiten Abschnitt wird dann gezeigt, wie sie genutzt werden können, um auf die Properties resp. Felder von Entity-Objekten lesend resp. schreibend zuzugreifen.

15.1 EntityType

Im folgenden wird die Verwendung der JPA-Klassen `Metamodel`, `EntityType` und `Attribute` genutzt werden können, um die JPA-relevanten Informationen aus einer `@Entity`-Klasse auslesen zu können.

Wie benutzen die bekannten Klasse `Book` und `Publisher` (zwischen denen eine `@ManyToOne`-Beziehung existiert).

Application

```
package appl;
// ...
import javax.persistence.metamodel.Attribute;
import javax.persistence.metamodel.EntityType;
import javax.persistence.metamodel.Metamodel;

public class Application {

    public static void main(String[] args) { ... }

    static void demo1(TransactionTemplate tt) {
        Util.mlog();
        tt.run(manager -> {
            final Metamodel model =
                manager.getEntityManagerFactory().getMetamodel();
            final EntityType<Book> bookType =
                model.entity(Book.class);
            final EntityType<Publisher> publisherType =
                model.entity(Publisher.class);
            showEntityType(bookType);
            showEntityType(publisherType);
        });
    }

    static void demo2(TransactionTemplate tt) {
        Util.mlog();
        tt.run(manager -> {
            final Metamodel model =
                manager.getEntityManagerFactory().getMetamodel();
            final Set<EntityType<?>> types = model.getEntities();
            types.forEach(Application::showEntityType);
        });
    }

    private static void showEntityType(EntityType type) {
        final Class<?> cls = type.getJavaType();
        System.out.println("Class: " + cls.getName());
        final Set<Attribute> attributes = type.getAttributes();
        attributes.forEach(attribute -> {
            System.out.println("\t" +
                attribute.getName());
        });
    }
}
```

```

        System.out.println("\t\t" +
            attribute.getJavaType());
        System.out.println("\t\t" +
            attribute.getJavaMember());
        System.out.println("\t\t" +
            attribute.getPersistentAttributeType());
    });
    System.out.println();
}
}

```

Die Ausgaben:

```

+-----+
| demol |
+-----+
Class: domain.Book
  id
    class java.lang.Integer
    public java.lang.Integer domain.Book.getId()
    BASIC
  isbn
    class java.lang.String
    public java.lang.String domain.Book.getIsbn()
    BASIC
  publisher
    class domain.Publisher
    public domain.Publisher domain.Book.getPublisher()
    MANY_TO_ONE
  title
    class java.lang.String
    public java.lang.String domain.Book.getTitle()
    BASIC
  price
    double
    public double domain.Book.getPrice()
    BASIC

Class: domain.Publisher
  books
    interface java.util.Collection
    private java.util.Collection domain.Publisher.books
    ONE_TO_MANY
  id
    class java.lang.Integer
    private java.lang.Integer domain.Publisher.id
    BASIC
  name
    class java.lang.String
    private java.lang.String domain.Publisher.name
    BASIC

```

```

+-----+
| demo2
+-----+
Class: domain.Book
    id
        class java.lang.Integer
        public java.lang.Integer domain.Book.getId()
        BASIC
    isbn
        class java.lang.String
        public java.lang.String domain.Book.getIsbn()
        BASIC
    publisher
        class domain.Publisher
        public domain.Publisher domain.Book.getPublisher()
        MANY_TO_ONE
    title
        class java.lang.String
        public java.lang.String domain.Book.getTitle()
        BASIC
    price
        double
        public double domain.Book.getPrice()
        BASIC

Class: domain.Publisher
    books
        interface java.util.Collection
        private java.util.Collection domain.Publisher.books
        ONE_TO_MANY
    id
        class java.lang.Integer
        private java.lang.Integer domain.Publisher.id
        BASIC
    name
        class java.lang.String
        private java.lang.String domain.Publisher.name
        BASIC

```

15.2 Access

Die JPA-Metainformationen können auch genutzt werden, um lesend und schreibend auf die Attribute / Properties eines `@Entity`-Objekts zuzugreifen. Im folgenden wird nur der lesende Zugriff vorgestellt:

Application

```
package appl;
// ...
import java.lang.reflect.Field;
import java.lang.reflect.Member;
import java.lang.reflect.Method;

import javax.persistence.metamodel.Attribute;
import javax.persistence.metamodel.EntityType;
import javax.persistence.metamodel.Metamodel;

public class Application {

    public static void main(String[] args) { ... }

    static void demoRead(TransactionTemplate tt) {
        Util.mlog();
        tt.run(manager -> {
            final Publisher publisher = new Publisher("Addison");
            final Book book =
                new Book("11111", "Pascal", 10, publisher);

            final Metamodel model =
                manager.getEntityManagerFactory().getMetamodel();
            final EntityType bookType = model.entity(Book.class);
            showEntity(bookType, book);
            final EntityType publisherType =
                model.entity(Publisher.class);
            showEntity(publisherType, publisher);
        });
    }

    static void showEntity(EntityType type, Object entity) {
        final Set<Attribute> attributes = type.getAttributes();
        System.out.println(type.getJavaType());
        attributes.forEach(attribute -> {
            final Member member = attribute.getJavaMember();
            final Object value;
            try {
                if (member instanceof Method) {
                    final Method method = (Method) member;
                    method.setAccessible(true);
                    value = method.invoke(entity);
                }
                else {
                    final Field field = (Field) member;

```

```
        field.setAccessible(true);
        value = field.get(entity);
    }
    catch (final Exception e) {
        throw new RuntimeException(e);
    }
    System.out.println(
        "\t" + member.getName() + " => " + value);
    });
}
```

Die Ausgaben:

```
+-----+
| demoRead
+-----+
class domain.Book
    getPrice => 10.0
    getPublisher => Publisher [null, Addison]
    getIsbn => 11111
    getId => null
    getTitle => Pascal
class domain.Publisher
    books => [Book [null, 11111, 10.0, Pascal]]
    id => null
    name => Addison
```

16 Der Second-Level Cache

Jede `EntityManager`-Instanz besitzt bekanntlich ihren eigenen Cache – den "First-Level Cache". Darüber hinaus ist es auch möglich, einen weiteren Cache zu nutzen: den Second-Level Cache. Dieser ist mit der `EntityManagerFactory` verbunden. Er ist somit ebenso "global" wie die `EntityManagerFactory` und überdauert also die einzelnen Sessions. Um den Second-Level Cache nutzen zu können, muss dieser explizit registriert werden.

Dieser Second-Level Cache sollte nur dann genutzt werden, wenn er Daten enthält, welche nur selten geändert werden (er ist mit Vorsicht zu genießen...)

16.1 Read-Only-Zugriffe

Wir benötigen zunächst eine neue Datei (die im CLASSPATH liegen muss):

ehcache.xml

```
<ehcache>
  <defaultCache
    maxElementsInMemory="10000"
    eternal="true"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"
  />
</ehcache>
```

Die persistence.xml muss erweitert werden:

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence .../>

  <persistence-unit name="library">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>domain.Category</class>
    <properties>
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="false" />

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

    </properties>

  </persistence-unit>
</persistence>
```

Als beispielhafte persistente Klasse wird die Klasse `Category` genutzt (die `CATEGORY`-Tabelle wird nur selten geändert – hauptsächlich finden Lesezugriffe statt):

create.sql

```
create table Category (
    id integer generated by default as identity (start with 1),
    type varchar(255) not null,
    description varchar(255) not null,
    primary key (id),
    unique (type)
);
```

domain.Category

```
package domain;
// ...
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity

@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY)

public class Category {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Basic
    private String type;

    @Basic
    private String description;

    public Category() {
        System.out.println("Category.<init>");
    }

    public Category(String type, String description) { ... }

    // getter, setter, toString...
}
```

Hier die Beispiel-Applikation:

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) { ... }

    static void demoPersist(TransactionTemplate tt) {
        Util.mlog();
    }
}
```



```

        tt.run(manager -> {
            manager.persist(new Category("I", "Informatik"));
            manager.persist(new Category("B", "Belletristik"));
            manager.persist(new Category("M", "Mathematik"));
        });
    }

    static void demoFind(TransactionTemplate tt) {
        Util.mlog();
        for (int i = 0; i < 3; i++)
            tt.run(manager ->
                System.out.println(manager.find(Category.class, 1)));
    }

    static void demoQuery(TransactionTemplate tt) {
        Util.mlog();
        for (int i = 0; i < 3; i++) {
            tt.run(manager -> {
                final String jpql =
                    "select c from Category c where c.type = :type";
                final TypedQuery<Category> query =
                    manager.createQuery(jpql, Category.class);
                query.setParameter("type", "I");
                System.out.println(query.getSingleResult());
            });
        }
    }
}

```

Die Ausgaben:

```

Category.<init>
+-----+
| demoPersist
+-----+
Hibernate: insert into Category (...) values (default, ?, ?)
Hibernate: insert into Category (...) values (default, ?, ?)
Hibernate: insert into Category (...) values (default, ?, ?)
+-----+
| demoFind
+-----+
Hibernate: select ... from Category where id=?
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Informatik, 1, I]
+-----+
| demoQuery
+-----+
Hibernate: select ... from Category where type=?
Category.<init>

```

```
Category [Informatik, 1, I]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Informatik, 1, I]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Informatik, 1, I]
```

Werden in ein und derselben Transaktion mehrere `find`-Aufrufe mit derselben ID ausgeführt, so wird stets ein neues(!) Objekt zurückgeliefert (obgleich das Objekt auch im First-Level Cache vorhanden ist).

Werden in einer Transaktion mehrere Queries mit derselben Bedingung ausgeführt, so findet jedes Mal ein neuer `SELECT` statt.

16.2 Read-Write

domain.Category

```
package domain;
// ...
@Entity
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_WRITE)

public class Category {
    // ...
}
```

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) { ... }

    static void demoPersist(TransactionTemplate tt) {
        Util.mlog();
        tt.run(manager -> {
            manager.persist(new Category("I", "Informatik"));
            manager.persist(new Category("B", "Belletristik"));
            manager.persist(new Category("M", "Mathematik"));
        });
    }

    static void demoFindUpdateFind(TransactionTemplate tt) {
        Util.mlog();
        for (int i = 0; i < 3; i++)
            tt.run(manager ->
                System.out.println(manager.find(Category.class, 1)));
        tt.run(manager -> {
            final Category c = manager.find(Category.class, 1);
            final Category cat = manager.merge(c);
            cat.setDescription("Computer Science");
        });
        for (int i = 0; i < 3; i++)
            tt.run(manager ->
                System.out.println(manager.find(Category.class, 1)));
    }
}
```

Die Ausgaben:

Category.<init>

+-----

```
| demoPersist
+-----
Hibernate: insert into Category (...) values (default, ?, ?)
Hibernate: insert into Category (...) values (default, ?, ?)
Hibernate: insert into Category (...) values (default, ?, ?)
+-----
| demoFindUpdateFind
+-----
Hibernate: select ... from Category where id=?
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Computer Science, 1, I]
Category.<init>
Category [Computer Science, 1, I]
Category.<init>
Category [Computer Science, 1, I]
```

16.3 Query-Cache

domain.Category

```
package domain;
// ...
@Entity
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY)

public class Category {
    // ...
}
```

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) { ... }

    static void demoPersist(TransactionTemplate tt) { ... }

    static void demoQuery(TransactionTemplate tt) {
        Util.mlog();
        for (int i = 0; i < 3; i++) {
            tt.run(manager -> {

                final TypedQuery<Category> query =
                    manager.createQuery(
                        "select c from Category c where c.type = :type",
                        Category.class);

                query.setHint(QueryHints.HINT_CACHEABLE, true);

                query.setParameter("type", "I");
                System.out.println(query.getSingleResult());

                query.setParameter("type", "B");
                System.out.println(query.getSingleResult());

                query.setParameter("type", "M");
                System.out.println(query.getSingleResult());

            });
        }
    }
}
```

Die Ausgaben:

+-----

```
| demoQuery
+-----
Hibernate: select ... from Category where type=?
Category.<init>
Category [Informatik, 1, I]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Belletristik, 2, B]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Mathematik, 3, M]
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Belletristik, 2, B]
Category.<init>
Category [Mathematik, 3, M]
Category.<init>
Category [Informatik, 1, I]
Category.<init>
Category [Belletristik, 2, B]
Category.<init>
Category [Mathematik, 3, M]
```

16.4 Regions

Im Second-Level Cache können mehrere Regionen (mit unterschiedlichen Konfigurationen) eingerichtet werden:

ehcache.xml

```
<ehcache>
  <defaultCache
    maxElementsInMemory="10000"
    eternal="true"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"
  />

  <cache name="cat"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="1"
    timeToLiveSeconds="1"
    overflowToDisk="true"
  />
</ehcache>
```

Der `cat`-Cache ist derart konfiguriert, dass die darin enthaltenen Objekte nur sehr kurzlebig sind...

domain.Category

```
package domain;
// ...
@Entity
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_ONLY, region="cat")

public class Category {
    // ...
}
```

appl.Application

```
package appl;
// ...
public class Application {

    public static void main(String[] args) { ... }

    static void demoPersist(TransactionTemplate tt) { ... }
```

```

static void demoQuery(TransactionTemplate tt) {
    Util.mlog();
    for (int i = 0; i < 3; i++) {
        Util.sleep(2_000);
        tt.run(manager -> {
            final TypedQuery<Category> query =
                manager.createQuery(
                    "select c from Category c where c.type = :type",
                    Category.class);

            query.setHint(QueryHints.HINT_CACHEABLE, true);
            query.setHint(QueryHints.HINT_CACHE_REGION, "cat");

            query.setParameter("type", "I");
            System.out.println(query.getSingleResult());

            query.setParameter("type", "B");
            System.out.println(query.getSingleResult());

            query.setParameter("type", "M");
            System.out.println(query.getSingleResult());
        });
    }
}

```

Man beachte, dass `demoQuery` vor jeder Transaktion 2 Sekunden schläft...

Die Ausgaben:

```

+-----+
| demoQuery
+-----+
Hibernate: select ... from Category where type=?
Category.<init>
Category [Informatik, 1, I]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Belletristik, 2, B]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Mathematik, 3, M]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Informatik, 1, I]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Belletristik, 2, B]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Mathematik, 3, M]
Hibernate: select ... from Category where type=?
Category.<init>

```

```
Category [Informatik, 1, I]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Belletristik, 2, B]
Hibernate: select ... from Category where type=?
Category.<init>
Category [Mathematik, 3, M]
```

17 Environments

Im folgenden wird anhand zweier einfacher Beispiele der Einsatz von JPA in einer Servlet- resp. einer EJB-Umgebung demonstriert.

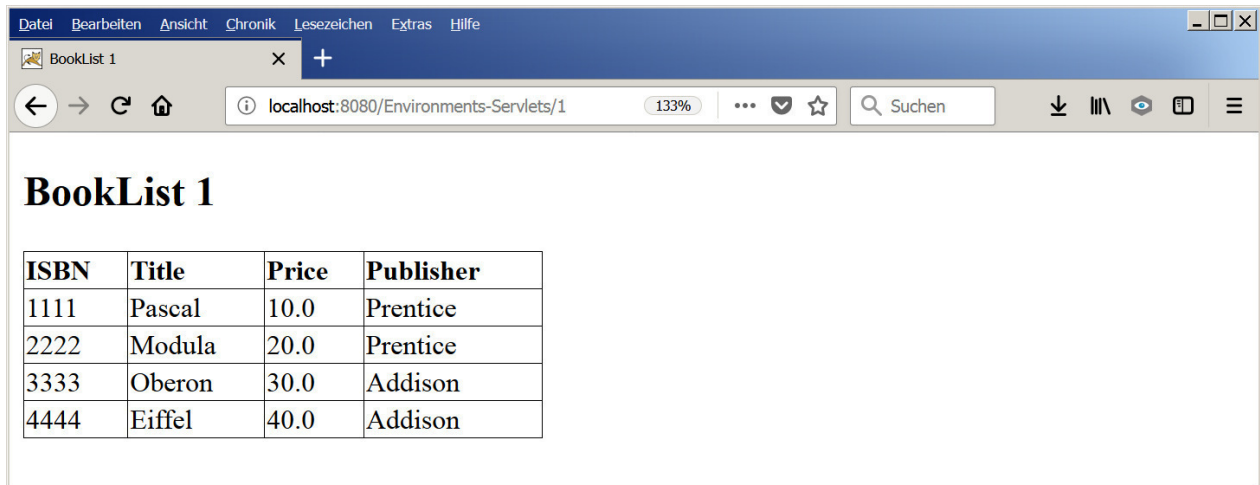
Im ersten Beispiel geht's um Servlets. Als Servlet-Container wird Tomcat verwendet.

Im zweiten Beispiel geht's um EJBs. Als EJB-Container wird JBoss verwendet.

In beiden Beispielen wird die HSQLDB verwendet.

17.1 Servlets

Die Oberfläche präsentiert sich wie folgt:



Nach dem Applikationsnamen kann als Servlet-Path 1, 2 oder 3 eingegeben werden. Es erscheint (bis auf die Überschrift) immer dasselbe Bild.

Die Anwendung wird mit `ant` gebaut und deployt.

Um die Anwendung zu erstellen, muss zunächst das `_install`-Target der `build.xml` ausgeführt werden. Für das eigentliche Deployment ist das `deploy`-Target zuständig.

Als persistenten Klassen werden `Book` und `Publisher` verwendet (mit einer `@ManyToMany`-Assoziation).

Die Datenbanktabellen werden aufgrund folgender Anweisungen erzeugt:

create.sql

```
create table PUBLISHER (  
    // ...  
);  
  
create table BOOK (  
    // ...  
);  
  
insert into publisher (name) values('Prentice');  
insert into publisher (name) values('Addison');  
insert into book (isbn, title, price, publisher_id)  
    values('1111', 'Pascal', 10.0, 1);  
insert into book (isbn, title, price, publisher_id)  
    values('2222', 'Modula', 20.0, 1);
```

```
insert into book (isbn, title, price, publisher_id)
  values('3333', 'Oberon', 30.0, 2);
insert into book (isbn, title, price, publisher_id)
  values('4444', 'Eiffel', 40.0, 2);
```

Um die automatische Erstellung der Datenbank aufgrund der `create.sql`-Datei kümmert sich ein `DbServletContextListener`:

DbServletContextListener

```
package util;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import db.util.batch.Executor;
import db.util.logger.PrintStreamLogger;

public class DbServletContextListener implements ServletContextListener {

    public static final String DB_PROPERTIES = "db.properties";
    public static final String CREATE_SQL = "create.sql";

    @Override
    public void contextInitialized(ServletContextEvent event) {
        Executor.execute(DB_PROPERTIES, "prepare", CREATE_SQL,
            new PrintStreamLogger(System.err));
        Executor.execute(DB_PROPERTIES, "select", null,
            new PrintStreamLogger(System.err));
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        Executor.execute(DB_PROPERTIES, "select", null,
            new PrintStreamLogger(System.err));
    }
}
```

Dieser `ServletContextListener` wird in mittels der `web.xml` registriert werden (s.u.). Man beachte hier die Benutzung der Klassen des `db-util`-Projekts.

Zur Infrastruktur gehört schließlich ein weiterer `ServletContextListener`:

JPAServletContextListener

```
package util;

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
```

```

public class JPAServletContextListener implements ServletContextListener {

    public final static String ENTITY_MANAGER_FACTORY =
        "EntityManagerFactory";

    public static EntityManagerFactory getEntityManagerFactory(
        ServletContext context) {
        final EntityManagerFactory factory =
            (EntityManagerFactory) context
                .getAttribute(ENTITY_MANAGER_FACTORY);
        if (factory == null)
            throw new RuntimeException(
                "EntityManagerFactory not registered - ...");
        return factory;
    }

    @Override
    public void contextInitialized(ServletContextEvent event) {
        final EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("library");
        event.getServletContext().setAttribute(
            ENTITY_MANAGER_FACTORY, factory);
        System.out.println(
            "EntityManagerFactory created and registered!");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        final EntityManagerFactory factory =
            getEntityManagerFactory(event.getServletContext());
        factory.close();
        event.getServletContext()
            .removeAttribute(ENTITY_MANAGER_FACTORY);
        System.out.println(
            "EntityManagerFactory closed and deregistered!");
    }
}

```

Mittels dieses Listeners wird die `EntityManagerFactory` erzeugt und im `ServletContext` registriert resp. geschlossen und deregistriert. Mittels des Aufrufs der statischen Methode `getEntityManagerFactory` kann die Factory überall ermittelt werden, wo sie benötigt wird.

Soviel zur Infrastruktur. Nun zur eigentlichen Anwendung. Zuerst ein (sehr einfaches!) Interface und seine Implementierung:

LibraryService

```

package services;
// ...
public interface LibraryService {
    public abstract List<Book> getBookListWithPublisher();
}

```

LibraryServiceImpl

```
package services;
// ...
public class LibraryServiceImpl implements LibraryService {

    private EntityManager manager;

    public void setEntityManager(EntityManager manager) {
        this.manager = manager;
    }

    @Override
    public List<Book> getBookListWithPublisher() {
        final String jpql =
            "select b from Book b join fetch b.publisher";
        return this.manager.createQuery(jpql).getResultList();
    }
}
```

Per `setEntityManager` wird ein `DelegatingEntityManager` injiziert werden Dieser delegiert dann an den mit dem jeweiligen Request (also mit dem jeweiligen Thread) assoziierten "richtigen" `EntityManager`.

Mittels eines `ServiceRegistrationListeners` werden die verfügbaren Service-Objekte im `ServletContext` registriert:

ServiceRegistrationListener

```
package listeners;
// ...
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class ServiceRegistrationListener implements ServletContextListener {

    public final static String LIBRARY_SERVICE = "LibraryService";

    public static LibraryService getLibraryService(ServletContext context) {
        final LibraryService service =
            (LibraryService)context.getAttribute(LIBRARY_SERVICE);
        if (service == null)
            throw new RuntimeException(
                "LibraryService not registered - ...");
        return service;
    }

    @Override
    public void contextInitialized(ServletContextEvent event) {
        final EntityManagerFactory factory =
            JPAServletContextListener
```

```

        .getEntityManagerFactory(event.getServletContext());
final LibraryServiceImpl serviceImpl =
    new LibraryServiceImpl();
serviceImpl.setEntityManager(
    DelegatingEntityManager.getInstance());
final LibraryService service =
    TransactionHandler.createProxy(
        factory, LibraryService.class, serviceImpl);
event.getServletContext()
    .setAttribute(LIBRARY_SERVICE, service);
System.out.println("Services created and registered!");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
    }
}

```

Mittels der statischen Methode `getLibraryService` kann der `LibraryService` überall ermittelt werden, wo er benötigt wird. Die Implementierungs-Klasse muss dem Nutzer dieses Services nicht bekannt sein.

Schließlich enthält die Anwendung drei verschiedene Servlet-Klassen, die jeweils ein anderes Herangehen an JPA demonstrieren (diese werden mit dem ServletPath 1, 2 bzw. 3 aufgerufen – s.o.).

Die erste Servlet-Klasse kümmert sich selbst um das erforderliche "Drumherum" (Erzeugung des `EntityManagers` und der Transaktion; Commit der Transaktion und Schließen des `EntityManagers` etc.):

LibraryServlet1

```

package servlets;
// ...
public class LibraryServlet1 extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    public void service(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        List<Book> bookList;

        final EntityManagerFactory factory =
            JPAServletContextListener.getEntityManagerFactory(
                this.getServletContext());
        final EntityManager manager =
            factory.createEntityManager();
        final EntityTransaction transaction =

```

```

        manager.getTransaction();
    try {
        transaction.begin();
        final String jpql =
            "select b from Book b join fetch b.publisher";
        bookList =
            manager.createQuery(jpql, Book.class).getResultList();
        transaction.commit();
    }
    catch (final RuntimeException e) {
        if (transaction.isActive())
            transaction.rollback();
        throw e;
    }
    finally {
        manager.close();
    }

    request.setAttribute("version", 1);
    request.setAttribute("bookList", bookList);
    request.getRequestDispatcher("WEB-INF/jsp/BookList.jsp")
        .forward(request, response);
}
}

```

Die zweite Servlet-Klasse benutzt das `TransactionTemplate`:

LibraryServlet2

```

package servlets;
// ...
public class LibraryServlet2 extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    public void service(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        final EntityManagerFactory factory =
            JPAServletContextListener
                .getEntityManagerFactory(this.getServletContext());
        final TransactionTemplate tt =
            new TransactionTemplate(factory);
        final List<Book> bookList = tt.runWithResult(manager -> {
            final String jpql =
                "select b from Book b join fetch b.publisher";
            return manager
                .createQuery(jpql, Book.class).getResultList();
        });

        request.setAttribute("version", 2);
        request.setAttribute("bookList", bookList);
        request.getRequestDispatcher("WEB-INF/jsp/BookList.jsp")

```



```

        .forward(request, response);
    }
}

```

Die dritte Klasse schließlich benutzt Dynamic-Proxies (und ist naturgemäß die kürzeste):

LibraryServlet3

```

package servlets;
// ...
public class LibraryServlet3 extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    public void service(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        LibraryService service = ServiceRegistrationListener
            .getLibraryService(this.getServletContext());
        List<Book> bookList = service.getBookListWithPublisher();

        request.setAttribute("version", 3);
        request.setAttribute("bookList", bookList);
        request.getRequestDispatcher("WEB-INF/jsp/BookList.jsp")
            .forward(request, response);
    }
}

```

Alle drei Servlet-Klassen nutzen als View-Implementierung eine JSP-Seite:

BookList.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<title>BookList ${version}</title>
<style type="text/css">
table {
    border-collapse: collapse;
}
</style>
</head>
<body>

<h2>BookList ${version}</h2>

```

```

<form>
<table border = "1" width="300">
  <tr>
    <th align="left">ISBN</th>
    <th align="left">Title</th>
    <th align="left">Price</th>
    <th align="left">Publisher</th>
  </tr>
  <c:forEach var="book" items="${bookList}">
    <tr>
      <td>${book.isbn}</td>
      <td>${book.title}</td>
      <td>${book.price}</td>
      <td>${book.publisher.name}</td>
    </tr>
  </c:forEach>
</table>
</form>

</body>
</html>

```

Zuletzt schließlich die `web.xml`:

web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>

  <listener>
    <listener-class>util.DbServletContextListener</listener-class>
  </listener>
  <listener>
    <listener-class>util.JPAServletContextListener</listener-class>
  </listener>
  <listener>
    <listener-class>listeners.ServiceRegistrationListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>LibraryServlet1</servlet-name>
    <servlet-class>servlets.LibraryServlet1</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>LibraryServlet2</servlet-name>
    <servlet-class>servlets.LibraryServlet2</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>LibraryServlet3</servlet-name>
    <servlet-class>servlets.LibraryServlet3</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>LibraryServlet1</servlet-name>
    <url-pattern>/1</url-pattern>
  </servlet-mapping>

```

```
</servlet-mapping>
<servlet-mapping>
  <servlet-name>LibraryServlet2</servlet-name>
  <url-pattern>/2</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>LibraryServlet3</servlet-name>
  <url-pattern>/3</url-pattern>
</servlet-mapping>
</web-app>
```

Ein letzter Hinweis zur Implementierung: Die Erzeugung der `EntityManagerFactory` und des Service-Objekts (resp. der Service-Objekte, wenn's denn mehrere wären) findet in `ServletContextListeners` statt. Die `init`-Methode der `Servlet`-Klasse ist hierfür NICHT geeignet – denn für jede der `Servlet`-Klassen würden dann jeweils neue Objekte erzeugt. Auch Singleton-Klassen sind für die Erzeugung zumindest des Service-Objekts (der Service-Objekte) nicht geeignet – müsste den `Servlets` doch dann der Name der Singleton-Klasse bekannt sein. Die Lösung mittels des Konzepts der `ServletContextListener` ist wahrscheinlich die bessere Variante.

17.2 EJB

Die Datenbank muss auch hier gestartet werden – und natürlich der JBoss (bin/standalone.bat) Das Deployment kann mit dem `deploy`-Target ausgeführt werden.

Die Dynamic-Proxy-Infrastruktur, die im letzten Beispiel (und auch in einigen Beispielen am Anfang dieses Skripts) entwickelt und benutzt wurde, ist in einem EJB-Kontext automatisch bereits vorhanden. Die vorliegende EJB-Anwendung kommt also ganz ohne "eigene" Infrastruktur-Klasse aus.

Auch dieses Beispiel verwendet wieder die persistenten Klassen `Publisher` und `Book` (wie auch im letzten Servlet-Beispiel). Sie sind nun allerdings in einem Package enthalten, welches sowohl serverseitig als auch clientseitig genutzt wird: dem Paket `common.domain`:

Publisher

```
package common.domain;
// ...
@Entity
public class Publisher {
    // ...
}
```

common.domain.Book

```
package common.domain;
// ...
@Entity
public class Book {
    // ...
}
```

Auch das Service-Interface sieht ähnlich aus wie das gleichnamige Interface, welches im Servlet-Abschnitt verwendet wurde (auch dieses liegt in einem `common....`-Paket, weil es sowohl vom Server als auch vom Client genutzt wird):

LibraryService

```
package common.services;

import javax.ejb.Remote;
// ...
@Remote
public interface LibraryService {
    public abstract List<Book> getBookListWithPublisher();
}
```

Der wichtige Unterschied zum Servlet-Beispiel besteht darin, dass dieses Interface mit `@Remote` annotiert ist.

Und hier die Implementierung (in einem Paket, welches nur auf der Server-Seite zugänglich ist):

LibraryServiceImpl

```
package server;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.hibernate.internal.SessionImpl;

import common.domain.Book;
import common.services.LibraryService;

@Stateless

// @TransactionManagement (TransactionManagementType.CONTAINER)
// this is the DEFAULT
// @TransactionAttribute(TransactionAttributeType.REQUIRED)
// this is the DEFAULT

public class LibraryServiceImpl implements LibraryService {

    private final boolean verbose = true;

    private EntityManager manager;

    @PersistenceContext
    public void setEntityManager(EntityManager manager) {
        if (this.verbose) {
            System.out.println("-----");
            System.out.println("==> setEntityManager");
            System.out.println("-----");
        }
        this.manager = manager;
    }

    @Override
    public List<Book> getBookListWithPublisher() {

        final SessionImpl session =
            (SessionImpl) this.manager.getDelegate();

        if (this.verbose) {
            System.out.println("-----");
            System.out.println("==> EntityManager          = " +
```

```

        this.manager);
        System.out.println("==> thread.id           = " +
            Thread.currentThread().getId());
        System.out.println("==> addr(session)         = " +
            System.identityHashCode(session));
        System.out.println("==> addr(session.transaction) = " +
            System.identityHashCode(session.getTransaction()));
        System.out.println("-----");
    }

    final String jpql =
        "select b from Book b join fetch b.publisher";
    return this.manager
        .createQuery(jpql, Book.class)
        .getResultList();
}
}

```

Die Service-Implementierung ist eine "Stateless Session Bean", eine Eigenschaft, welche mittels `@Stateless` annotiert wird.

Typischerweise kümmert sich der EJB-Container um das Handling der Transaktionen. Dieses Verhalten kann auch explizit gemacht werden:

```
@TransactionManagement(TransactionManagementType.CONTAINER)
```

(Die Alternative hierzu ist der `TransactionManagementType.BEAN` – dann muss die Bean sog. User-Transaktionen nutzen. Davon wird aber in aller Regel abgeraten.)

Per Default wird vom EJB-Container eine Transaktion erzeugt, bevor eine der Bean-Methoden aufgerufen wird. Nach Verlassen dieser Methode wird die Transaktion commitet (resp. zurückgesetzt – dann nämlich, wenn die Bean z.B. eine `RuntimeException` wirft – nähere Einzelheiten hierzu würden das vorliegende Skript sprengen). Auch hier kann dieses Default-Verhalten explizit gefordert werden:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
```

(Die Alternativen zu `REQUIRED` sollen hier nicht weiter dargestellt werden – aus dem oben genannten Grunde.)

Die Methode `setEntityManager` ist mittels `@PersistenceContext` annotiert:

```

private EntityManager manager;

@PersistenceContext
public void setEntityManager(EntityManager manager) {
    // ...
    this.manager = manager;
}

```

Der EJB-Container wird diese Methode aufrufen, um ein `EntityManager-Proxy` zu injizieren, welcher den Zugriff auf den mit dem aktuellen Thread assoziierten "richtigen" `EntityManager` vermittelt – ein ähnliches Konzept wie bei "unserem" `DelegatingEntityManager`.

Abgesehen von den ganzen Traces ist die Implementierung der Interface-Methode `getBookListWithPublisher` recht schlicht:

```
final String jpql =
    "select b from Book b join fetch b.publisher";
return this.manager
    .createQuery(jpql, Book.class)
    .getResultList();
```

Das heißt aber nicht, dass die Trace-Ausgaben nicht interessant wären. Ihre Interpretation sei dem Leser / der Leserin überlassen...

Hier schließlich der Client – eine einfache Console-Anwendung (implementiert in einem `client`-Paket, welche nur clientseitig verfügbar ist):

client.Client

```
package client;
// ...
import javax.naming.InitialContext;

public class Client {
    public static void main(String[] args) throws Exception {
        Db.aroundAppl();

        final InitialContext ctx = new InitialContext();

        final String jndiName =
            "ejb:x1402-Environments-EJB/x1402-Environments-" +
            "EJB/LibraryServiceImpl!" +
            LibraryService.class.getName();

        final LibraryService service =
            (LibraryService) ctx.lookup(jndiName);

        List<Book> bookList = service.getBookListWithPublisher();
        bookList = service.getBookListWithPublisher();
        bookList = service.getBookListWithPublisher();
        bookList.forEach(
            book -> System.out.println(
                book + " " + book.getPublisher()));
    }
}
```

Das Resultat des lookups ist ein Dynamic-Proxy, welches das Interface `LibraryService` implementiert. Der daran angeschlossene `InvocationHandler` kümmert sich dann um die Serialisierung des jeweiligen Methodenaufrufs an den Server – und natürlich um die Deserialisierung des vom Server gelieferten Ergebnisses.

Man beachte den dreimaligen Aufruf von `getBookListWithPublisher`. Das ist natürlich eigentlich unsinnig – es hat aber Bedeutung im Hinblick auf die Interpretation der serverseitigen Trace-Ausgaben...

Hier die zusätzlichen Konfigurations-Dateien, welche für die vorliegende Anwendung erforderlich sind:

create.sql

Es handelt sich hier um dieselbe Datei, welche auch im Servlet-Abschnitt verwendet wurde.

persistence.xml

```
<?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" version="1.0">

  <persistence-unit name="library">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <jta-data-source>java:/SEMINAR-DS</jta-data-source>

    <class>common.domain.Book</class>
    <class>common.domain.Publisher</class>

    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="false" />
    </properties>

  </persistence-unit>
</persistence>
```

In dieser `persistence.xml` wird die Datenbankverbindung nicht per expliziter Treiber-Klasse und URL bekanntgemacht, sondern über JNDI. Es wird eine JTA-Datenquelle verwendet, welche vom EJB-Container zur Verfügung und verwaltet wird. Der "logische" Name ist `java:/SEMINAR-DS`. Dieser Name wird in der folgenden Datei auf "physische" Namen abgebildet:

wildfly-8.2.0.Final\standalone\configuration\standalone.xml

Und schließlich sind clientseitig folgende Dateien erforderlich:

jndi.properties

```
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
```

jboss-ejb-client.properties

```
remote.connections=default  
remote.connection.default.host=127.0.0.1  
remote.connection.default.port=8080
```

18 Übungsaufgaben

Im den Übungen kann eine kleine Konto-Anwendung schrittweise entwickelt werden. Dabei werden diejenigen Klassen verwendet, welche im "Framework"-Kapitel vorgestellt wurden (`Configuration`, `TransactionHandler`, `DelegatingEntityManager`, `EntityManagerThreadLocal`).

18.1 Start

Voraussetzung: Kapitel "Basics" und "Framework"

Erstellen Sie eine persistente Klasse `Account`. Diese soll folgende Attribute / Properties besitzen: `id`, `number`, `number`, `balance`.

Erstellen Sie dann einen `AccountService` (Interface & Implementierung). Über das Interface sollen folgende Operationen erfolgen können:

```
createAccount  
findAccountByNumber  
findAllAccounts  
deposit  
withdraw
```

Definieren Sie (nur in der Implementierungs-Klasse!) eine `setEntityManager`-Methode, über welche die Injection des `EntityManagers` möglich ist.

Schreiben Sie eine kleine Anwendung, die den Service über ein Dynamic-Proxy nutzt (`TransactionHandler`!).

18.2 ServiceDao

Der Service soll nicht mehr direkt auf JPA zugreifen, sondern über ein DAO. Definieren Sie ein Interface `AccountDao` und eine Klasse `AccountDaoImpl`. Über `AccountDao` sollen folgenden Operationen erfolgen können:

```
findAccountByNumber  
findAllAccounts  
insert
```

Ändern Sie die `AccountService`-Implementierung derart, dass sie das `AccountDao` nutzt. Dabei sollte der Service nur vom Interface `AccountDao` abhängig sein, nicht aber von der Implementierungsklasse!

18.3 ManyToOne

Voraussetzung: Kapitel "Associations"

Definieren Sie die persistente Klasse `Customer` mit den folgenden Attributen / Properties: `id` und `name`.

Erweitern Sie die Klasse `Account` um ein Attribut namens `customer` (vom Typ `Customer`). Implementierung Sie also eine unidirektionale Assoziation von `Account` nach `Customer`

Erweitern Sie die `AccountService` und `AccountDao` um folgende Methoden:

```
findCustomerByName  
findAccountWithCustomerByNumber  
findAllAccountsWithCustomer
```

Erweitern Sie die Test-Anwendung!

18.4 OneToMany

Voraussetzung: Kapitel "Associations"

Ein Konto hat Bewegungen. Schreiben Sie eine persistente Klasse `Movement` mit den folgenden Attributen / Properties: `id`, `date`, `amount` und `account` (letztere vom Typ `Account`).

Erweitern Sie die Klasse `Account` um ein Attribut `movements` (vom Typ `List<Movement>`). Implementieren Sie also eine bidirektionale Beziehung zwischen `Account` und `Movement`.

Erweitern Sie die Methoden `deposit` und `withdraw` (hier müssen `Movements` erzeugt werden).

Erweitern Sie natürlich auch die Test-Anwendung!

18.5 Inheritance

Voraussetzung: Kapitel "Inheritance"

Benutzen Sie `Movement` als abstrakte Basisklasse, von welche z.B. `Deposit` und `Withdrawal` abgeleitet sind. Erzeugen Sie in der `deposit`-Methode dann `Deposit`-Objekte und in der `withdraw`-Methode `Withdrawal`-Objekte. Das `amount`-Attribut von `Movement` soll nun immer positiv sein!

Welche weiteren Möglichkeiten der Ableitung von `Movement` sind denkbar? Implementieren Sie eine dieser Möglichkeiten.

19 Literatur

Bauer, King: Java-Persistence mit Hibernate, Manning 2007

Bauer, King: Hibernate in Action, Manning 2004

Burke, Monson-Haefel: Enterprise JavaBeans 3.0, O'Reilly 2006