Java Persistence

Datenzugriffskomponente mit JPA 2.1

(Grundlagen der Java Persistence Architecture)

Vladislav Faerman

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN MÜNCHEN

JPA 2.1

Gliederung

- Einführung
- Konfiguration
- Objekt-Relationales Mapping (ORM) mit JPA
- Das zentrale Konzept der JPA
- Kaskadierung
- JPQL
- LiveCoding
- Zusammenfassung

Gliederung

- **Einführung**
- **Konfiguration**
- **Objekt-Relationales Mapping (ORM) mit JPA**
- **Das zentrale Konzept der JPA**
- **Kaskadierung**
- **JPQL**
- **Zusammenfassung**

JPA 2.1

Einführung zur Java Persistence API (JPA)

- **JPA** ist ein Standard für Objekt-Relationales Mapping (ORM) von Java Objekten
- Ziel:
  - Persistente Abspeicherung von Objekten in Datenbanken
- Hibernate, EclipseLink, Oracle TopLink Essentials etc. sind bekannte JPA- Implementierungen

JPA 2.1

Gliederung

- Einführung
- **Konfiguration**
- Objekt-Relationales Mapping (ORM) mit JPA
- Das zentrale Konzept der JPA
- Kaskadierung
- JPQL
- Zusammenfassung

JPA 2.1

Konfiguration

Einbindung von JPA in Java App mit Maven

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>edu.hm.maven</groupId>
	<artifactId>firstAppWithJPA</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<dependencies>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-entitymanager</artifactId>
			<version>4.3.6.Final</version>
		</dependency>
		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<version>5.1.31</version>
		</dependency>
	</dependencies>
</project>
```

JPA 2.1

Konfiguration der **persistence.xml** Verbindung zur Datenbank

```xml
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
version="2.1">
  <persistence-unit name="jpa-example" transaction-type="RESOURCE_LOCAL">
     <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
       <properties>
         <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/MySQLForJPA" />
         <property name="javax.persistence.jdbc.user" value="friendOfJPA" />
         <property name="javax.persistence.jdbc.password" value="password" />
         <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />

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

         <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
         <property name="hibernate.hbm2ddl.auto" value="create" />


         <!-- Configuring Connection Pool -->
         <property name="hibernate.c3p0.min_size" value="5" />
         <property name="hibernate.c3p0.max_size" value="20" />
         <property name="hibernate.c3p0.timeout" value="500" />
         <property name="hibernate.c3p0.max_statements" value="50" />
         <property name="hibernate.c3p0.idle_test_period" value="2000" />
       </properties>
  </persistence-unit>
</persistence>
```

A

B

JPA 2.1

Bedeutung von Property „hibernate.hbm2ddl.auto“

Wert	Auswirkung
validate	Validierung des Schemas, aber keine Veränderung an der Datenbank. Wenn das Datenbankschema zu dem aktuellen Mapping nicht kompatibel ist, wird eine Fehlermeldung produziert.
update	Falls das Datenbankschema zu dem aktuellen Mapping nicht kompatibel ist, werden die betroffenen Tabellen angepasst. Die Daten in den veränderten Tabellen können dadurch verloren gehen.
create	Beim Start der Anwendung werden alle Tabellen in der Datenbank gelöscht und das aktuelle Schema neuangelegt. Wenn die Anwendung herunterfährt, bleibt das Datenbankschema in der Datenbank.
create-drop	Beim Start der Anwendung werden alle Tabellen in der Datenbank gelöscht und das aktuelle Schema neuangelegt. Wenn die Anwendung herunterfährt, werden alle Tabellen in der Datenbank gelöscht.

JPA 2.1

Gliederung

- Einführung
- Konfiguration
- **Objekt-Relationales Mapping (ORM) mit JPA**
- Das zentrale Konzept der JPA
- Kaskadierung
- JPQL
- Zusammenfassung

JPA 2.1

Objekt-Relationales Mapping (ORM)

- Ein Verfahren zur Speicherung von Objekten in Datenbanken
- Die Klassenstruktur wird auf einer relationalen Datenbankschema abgebildet

Entitäten und die Beziehungen zwischen Entitäten

JPA 2.1

- Entität:
  - ein eindeutig identifizierendes Objekt
  - Beschreibung durch Eigenschaften
- Beziehungen:
  - 1:1- Beziehung
  - 1:n- Beziehung
  - n:1- Beziehung
  - n:m- Beziehung

JPA 2.1

Beziehungen zwischen Entitäten

- Die Beziehungen unterscheiden sich zwischen:
  - unidirektional
    - Verbindung nur von einem der beteiligten Entitäten
  - bidirektional
    - Verbindung von beiden beteiligten Entitäten

→ keine Auswirkung auf die Tabellenstruktur

JPA 2.1

Mapping einer Klasse

- Wichtige Annotationen für Mapping einer Klasse:
  - @Entity:
    - für Mapping einer best. Klasse zu einer Tabelle
  - @Id:
    - als Primärschlüssel für ein best. Attribut
  - @GeneratedValue
    - zusätzlich zur @Id- Annotation
    - Generator für Primärschlüssel

JPA 2.1

Mapping einer Klasse

Keine persistente Abspeicherung der Attribute gewünscht:

- die Annotation @Transient

P. S. Alle JPA- Annotationen sind im Package **javax.persistence.*** definiert

JPA 2.1

Unidirektionale 1:1-Beziehung

```java
@Entity
public class Auto {

	@Id @GeneratedValue
	private long id;

	private String marke;
	private String modell;
	private String kennzeichen;

	@OneToOne
	private Autobesitzer autoBesitzer;

	// Getter und Setter
}
```

```java
@Entity
public class Autobesitzer {

	@Id @GeneratedValue
	private long id;

	private String vorname;
	private String nachname;

	// Getter und Setter
}
```

Die Erweiterung der @OneToOne durch @JoinColumn-Annotation (optional):

```java
@OneToOne
@JoinColumn(name = "Autobesitzer")
private Autobesitzer autoBesitzer;
```

JPA 2.1

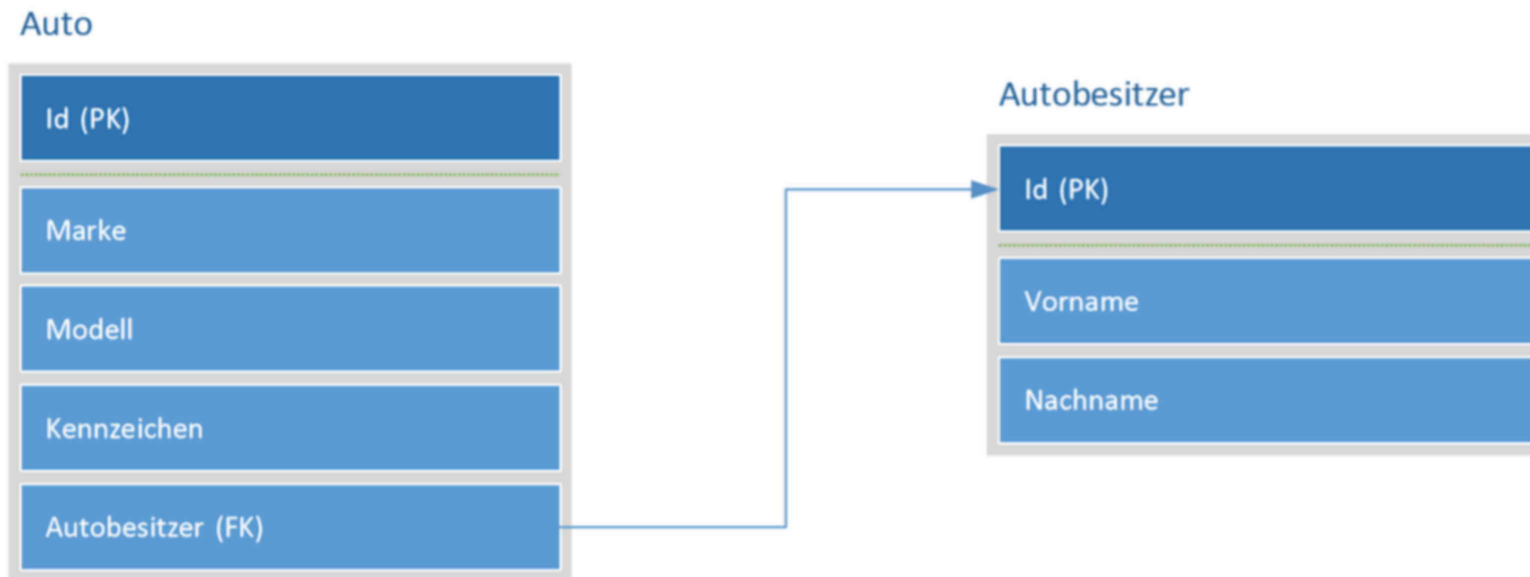
Bidirektionale 1:1-Beziehung

```
public class Auto {

        @Id @GeneratedValue
        private long id;

        private String marke;
        private String modell;
        private String kennzeichen;

        @OneToOne
        @JoinColumn(name = "Autobesitzer")
        private Autobesitzer autoBesitzer;

        // Getter und Setter
}
```

```
@Entity
public class Autobesitzer {

        @Id @GeneratedValue
        private long id;

        private String vorname;
        private String nachname;

        @OneToOne(mappedBy = "autoBesitzer")
        private Auto auto;

        // Getter und Setter
}
```

JPA 2.1

1:1- Beziehung
UML- Klassenmodell



JPA 2.1

1:n- Beziehung

- Erfolgt durch eine @OneToMany bzw. @ManyToOne- Annotation
- Auch hier sowohl bi- als auch unidirektionale Beziehung möglich
- Die bidirektionale Beziehung wird mittels mappedBy Attributs der @OneToMany- Annotation realisiert

JPA 2.1

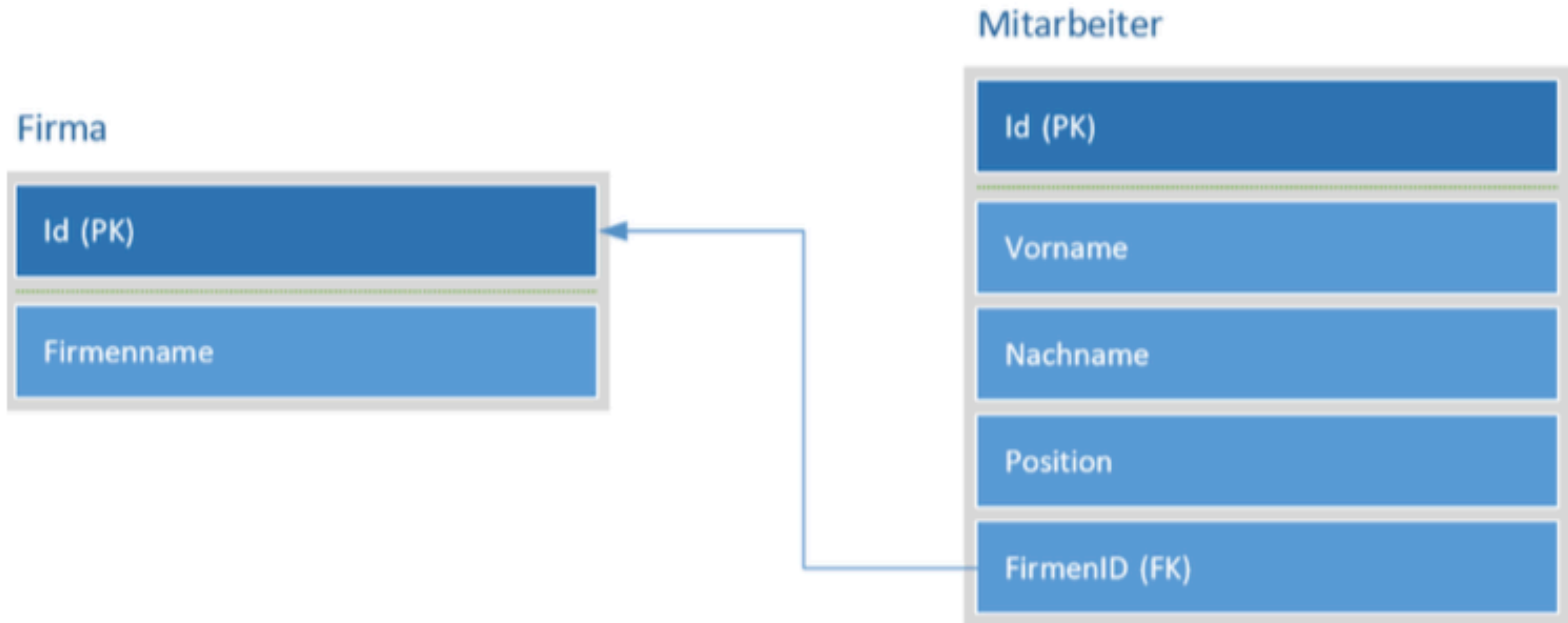
1:n- Beziehung

```
@Entity
public class Firma {

        @Id @GeneratedValue
        private long id;

        private String firmenName;

        @OneToMany(mappedBy = "firma")
        @JoinColumn(name = "FirmenID")
        private List<Mitarbeiter> mitarbeiters;

        // Getter und Setter
}
```

```
@Entity
public class Mitarbeiter {

        @Id @GeneratedValue
        private long id;

        private String vorname;
        private String nachname;
        private String position;

        @ManyToOne
        private Firma firma;

        // Getter und Setter
}
```

JPA 2.1

1:n- Beziehung
UML- Klassenmodell



JPA 2.1

n:m- Beziehung

- Erfolgt durch eine @ManyToMany- Annotation
- Solange eine unidirektionale n:m- Beziehung gewünscht ist, enthält nur die besitzende Seite die @ManyToMany- Annotation
- Bei der bidirektionalen Beziehung wird die @ManyToMany- Annotation auf beiden Seiten benutzt und mappedBy bei einer der Annotationen
- In der Datenbank wird automatisch eine Join- Tabelle mit Fremdschlüsseln angelegt

JPA 2.1

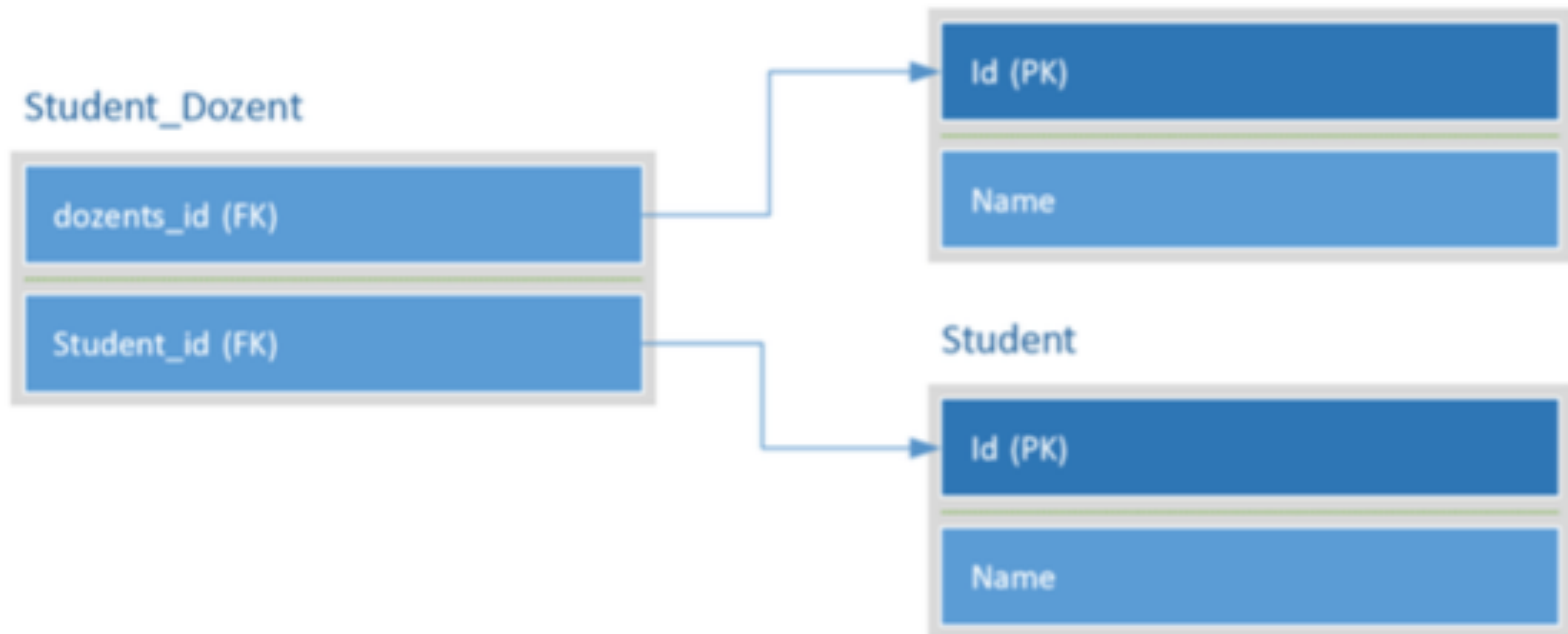
Unidirektionale n:m- Beziehung

```java
@Entity
public class Student {

	@Id @GeneratedValue
	private int id;
	private String name;

	@ManyToMany
	@JoinColumn(name = "Dozent")
	private List<Dozent> dozents;

	// Getter und Setter
}
```

```java
@Entity
public class Dozent {

	@Id @GeneratedValue
	private int id;
	private String name;

	// Getter und Setter
}
```

JPA 2.1

n:m- Beziehung
UML- Klassenmodell



- Ob es eine uni- bzw. bidirektionale Beziehung ist, hat das keine Auswirkung auf die Tabellenstruktur

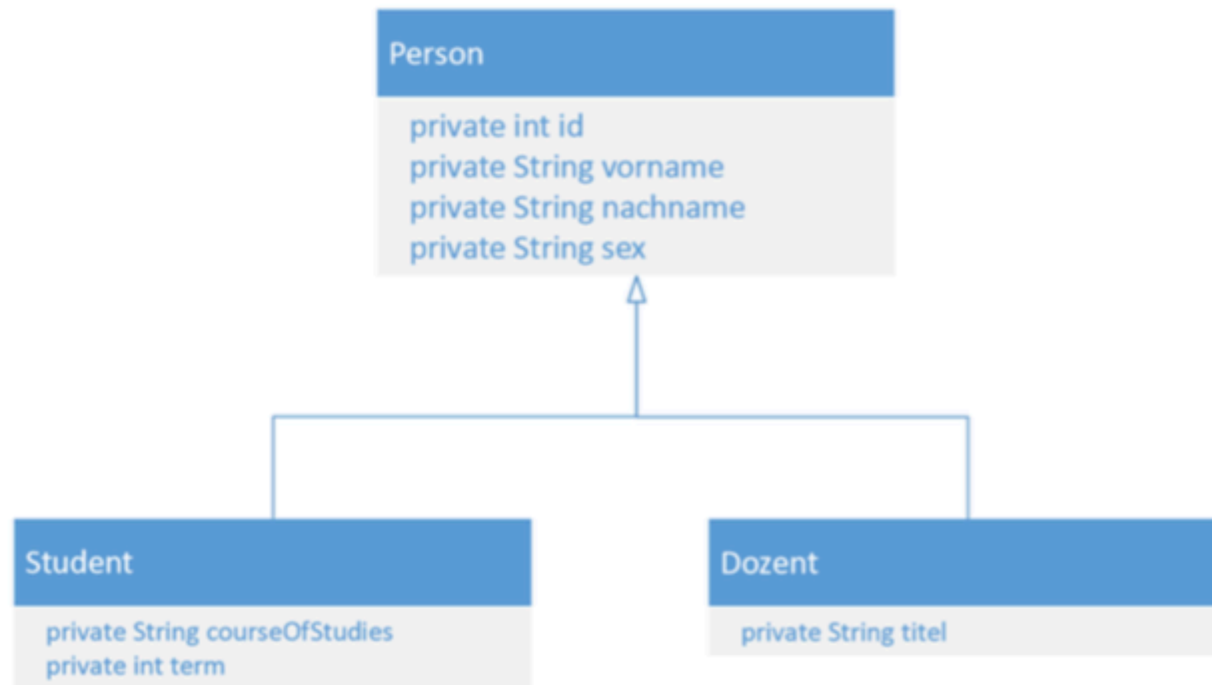
JPA 2.1

Vererbung

- Drei Strategien der Vererbung werden von JPA vorgesehen:
  - SINGLE_TABLE
  - JOINED
  - TABLE_PER_CLASS
- Umsetzung mithilfe der @Inheritance- Annotation

Verdeutlichung der Vererbung anhand eines Beispiels: „UML- Klassenmodell“

JPA 2.1



JPA 2.1

Vererbungsstrategie: „SINGLE TABLE“

- Erstellt eine einzige Tabelle für die gesamte Vererbungshierarchie
- S. g. Diskriminationsspalte wird automatisch für die Klassenzugehörigkeit des Objekts angelegt
- Die Angabe der Vererbungsstrategie ist in der obersten Basisklasse anzugeben

JPA 2.1

Oberste Basisklasse der Hierarchie „Person“

```
@Entity
@DiscriminatorColumn(name = "PersKlassen")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Person {

        @Id @GeneratedValue
        private long id;
        private String vorname;
        private String nachname;
        private String sex;

        // Getter und Setter
}
```

Unterklassen der Hierarchie

JPA 2.1

```java
@Entity
@DiscriminatorValue("Student")
public class Student extends Person {

	private String courseOfStudies;
	private int term;

	@ManyToMany
	private List<Dozent> dozents;

	 // Getter und Setter
}
```

```java
@Entity
@DiscriminatorValue("Dozent")
public class Dozent extends Person {

	private String titel;

	@ManyToMany(mappedBy = "dozents")
	private List<Student> students;

	 // Getter und Setter
}
```

Unterklassen definieren den Discriminator Value, d. h. Objekte nach Mapping werden in der Datenbank durch Discriminator Value konkretisiert

JPA 2.1

SINGLE TABLE:
Ergebnis in der Datenbank

PersKlassen	id	nachname	sex	vorname	titel	courseOfStudies	term
Dozent	1	Theis	männlich	Michael	Dozent	NULL	NULL
Student	2	Mustermann	männlich	Markus	NULL	Informatik	3

- Vorteil:
  - Performante Abfragen (keine Joins) → nur 1 Tabelle
- Nachteil:
  - je nach gespeicherten Objekt bleiben nicht benötigte Datenbankspalten leer (NOT NULL-Constraints nicht möglich)

JPA 2.1

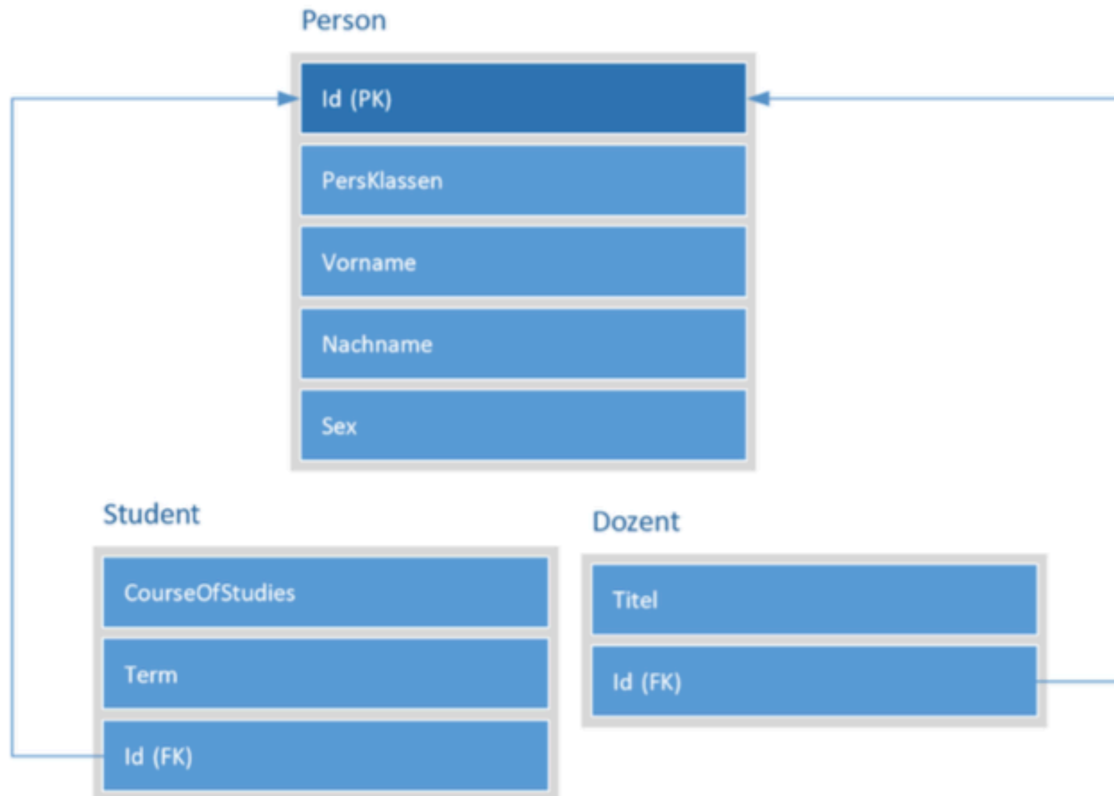
Vererbungsstrategie: „JOINED“

- Für jede konkrete und auch abstrakte Klasse wird jeweils eine Tabelle erstellt

- Jede Tabelle enthält Spalten nur für die Attribute, die direkt in der gemappten Klasse deklariert sind

JPA 2.1

UML- Klassenmodell für „JOINED“- Strategie



JPA 2.1

Vererbungsstrategie: „JOINED“

- Vorteil
  - keine null-Werte für nicht relevante Attribute

- Nachteil
  - Aufwändige Abfragen (Joins)

JPA 2.1

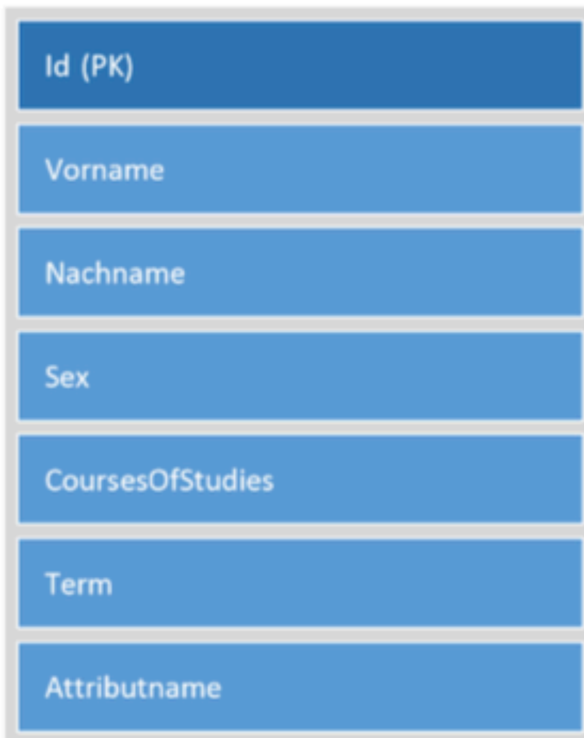
Vererbungsstrategie: „TABLE_PER_CLASS"

- Für jede NICHT abstrakte Klasse wird eine Tabelle angelegt
- Jede Tabelle enthält dann nach Mapping auch die Attribute der obersten Basisklasse
- Vorteile
  - Schnellere Abfrage der Objekte einer konkreten Klasse
- Nachteile
  - Aufwändige Abfragen (UNION)
  - Für polymorphe Queries muss jede betroffene Tabelle einzeln abgefragt werden
  - Keine expliziten Fremdschlüssel

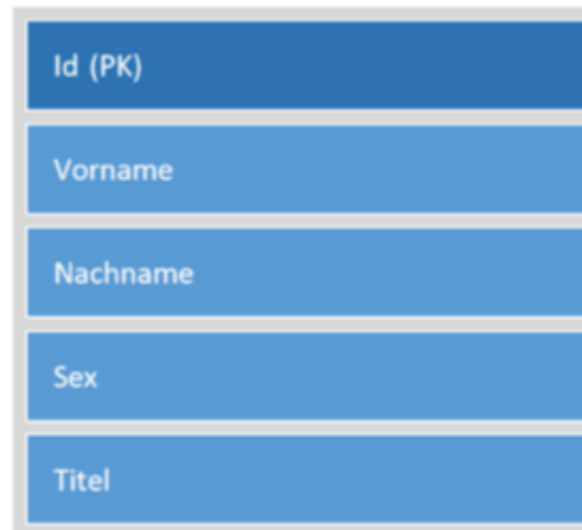
JPA 2.1

UML- Klassenmodell für „TABLE_PER_CLASS“- Strategie

Student



Dozent



Gliederung

- Einführung
- Konfiguration
- Objekt-Relationales Mapping (ORM) mit JPA
- **Das zentrale Konzept der JPA**
- Kaskadierung
- JPQL
- Zusammenfassung

JPA 2.1

Das zentrale Konzept der JPA

- **EntityManager** ist die zentrale Komponente in JPA
- Verwaltet die Schnittstelle zur Datenbank
- Alle Operationen mit den Entitäten werden durch EntityManager ausgeführt

JPA 2.1

Wichtige Methoden von EntityManager

Methoden	Beschreibung
persist()	Speichern eines Objektes in der Datenbank
remove()	Entfernen eines Objekts aus der Datenbank
find()	Finden eines Objekts aus der Datenbank über den Primärschlüssel (PK)
refresh()	Aktualisieren eines Objekts aus der Datenbank
createQuery()	Erzeugen einer Query mit JPQL
getTransaction()	gibt die aktuelle Transaktion zurück.

EntityManager

- EntityManager (*em*):
  - Transaktionen können durch EntityManager explizit gestartet und geschlossen werden:
    - *em.getTransaction().begin();*
      - ... DB-Kommunikation ...
    - *em.getTransaction().commit*
  - Innerhalb der laufenden Transaktion sind weitere Möglichkeiten wie Objekte speichern, ändern, löschen etc. möglich

Gliederung

JPA 2.1

- Einführung
- Konfiguration
- Objekt-Relationales Mapping (ORM) mit JPA
- Das zentrale Konzept der JPA
- **Kaskadierung**
- JPQL
- Zusammenfassung

Kaskadierung

JPA 2.1

- Für alle angesagten Annotationen wie *@OneToOne*, *@OneToMany* etc. kann auch s. g. Kaskadierung angewendet werden.

- Ziel ist:
  - Bei allen Operationen, die für Objekte durchzuführen sind, müssen auch für die referenzierten Objekte anderer Klassen durchgeführt werden

JPA 2.1

Beispiel 1 für die Kaskadierung: „CascadeType.ALL“

Entität-Klasse: Auto	Entität-Klasse: Autobesitzer

`@OneToOne(cascade = CascadeType.ALL)`
`@JoinColumn(name = "Autobesitzer")`
`private Autobesitzer autoBesitzer;`
	`@OneToOne(mappedBy = "autoBesitzer")`
	`@JoinColumn(name = "Auto")`
	`private Auto auto;`

- Alle Operationen, die mit Auto Objekt durchgeführt werden, werden mit dem referenzierten Autobesitzer Objekt auch durchgeführt

JPA 2.1

Beispiel für die Kaskadierung: „CascadeType.ALL“

```java
@Entity
public class Firma {

	@Id @GeneratedValue
	private long id;
	private String firmenName;

	@OneToMany(cascade = CascadeType.ALL)
	@JoinColumn(name = "FirmenID")
	private List<Mitarbeiter> mitarbeiters;

	// Getter und Setter
}
```

```java
@Entity
public class Mitarbeiter {

	@Id @GeneratedValue
	private long id;

	private String vorname;
	private String nachname;
	private String position;

	// Getter und Setter
}
```

- Alle Operationen, die mit Firma Objekt durchgeführt werden, werden auch mit dem referenzierten Mitarbeiter Objekt durchgeführt

JPA 2.1

Liste der Kaskadierungstypen

CascadeType[7]	Beschreibung
PERSIST	Beim Persistieren einer Entität werden referenzierte Entitäten gespeichert.
REMOVE	Beim Löschen einer Entität werden auch referenzierte Entitäten gelöscht.
REFRESH	Beim Aktualisieren einer Entität werden die referenzierten Entitäten aus der Datenbank erneut geladen.
MERGE	Bei der Merge- Operation werden die Änderungen an einer Entität in die Datenbank gespeichert. Für die referenzierten Entitäten bedeutet es, dass Ihre Änderungen auch in die Datenbank geschrieben werden.
ALL	Alle beschriebene Operationen werden auch für die referenzierte Entitäten durchgeführt.

JPA 2.1

Gliederung

- Einführung
- Konfiguration
- Objekt-Relationales Mapping (ORM) mit JPA
- Das zentrale Konzept der JPA
- Kaskadierung
- **JPQL**
- Zusammenfassung

JPA 2.1

JPQL
(Java Persistence Query Language)

- JPQL- eine Abfragesprache für die Entitäten
- Anlehnung an SQL
- JPQL- Abfragen werden von JPA in SQL übersetzt und Operationen werden in Datenbank ausgeführt
- *SELECT, UPDATE* und *DELETE* Operationen werden unterstützt

JPA 2.1

JPQL
Beispiel für select- Statement

- Beispiel für eine JPQL- Abfrage:

```
Query query = em.createQuery("select b from Autobesitzer b");
query.getResultList();
```

- Die Ergebnisliste *query* enthält Entities vom Typ Autobesitzer, die im Programm weiter verwendet können:

```
Hibernate:
  select
    autobesitz0_.id as id1_1_,
    autobesitz0_.nachname as nachname2_1_,
    autobesitz0_.vorname as vorname3_1_
  from
    Autobesitzer autobesitz0_
```

JPA 2.1

JPQL
Vergleich mit SQL

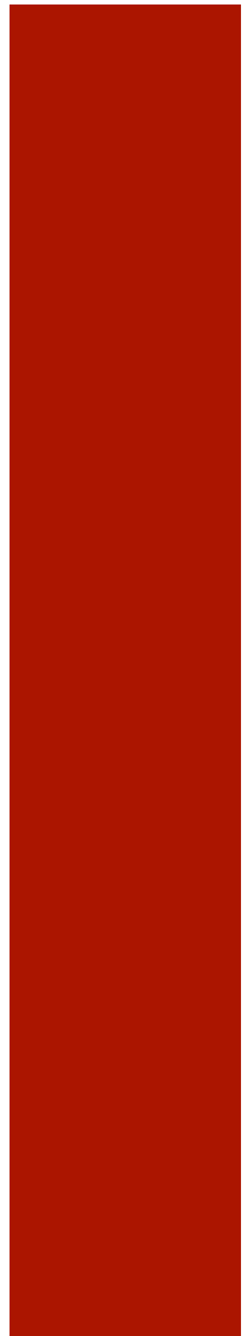
- JPQL *select*- Statement ist analog zu SQL aufgebaut:

  - *select*
  - *from*
  - *where*
  - *group by*
  - *having*
  - *order by*

- **Wichtiger Unterschied:**

  - Die Queries operieren mit Klassen und Klassenattributen und nicht mit Datensätzen

LiveCoding



JPA 2.1

Gliederung

- Einführung
- Konfiguration
- Objekt-Relationales Mapping (ORM) mit JPA
- Das zentrale Konzept der JPA
- Kaskadierung
- JPQL
- **Zusammenfassung**

JPA 2.1

Zusammenfassung

- Objekt-Relationales Mapping mit JPA und Hibernate-Implementierung erleichtert die Anwendungs-programmierung von Datenbankanwendungen

- Der Implementierungsaufwand für Projekte reduziert sich, da durch die Annotationen alle Tabellen und andere Datenbankobjekte automatisch erzeugt werden