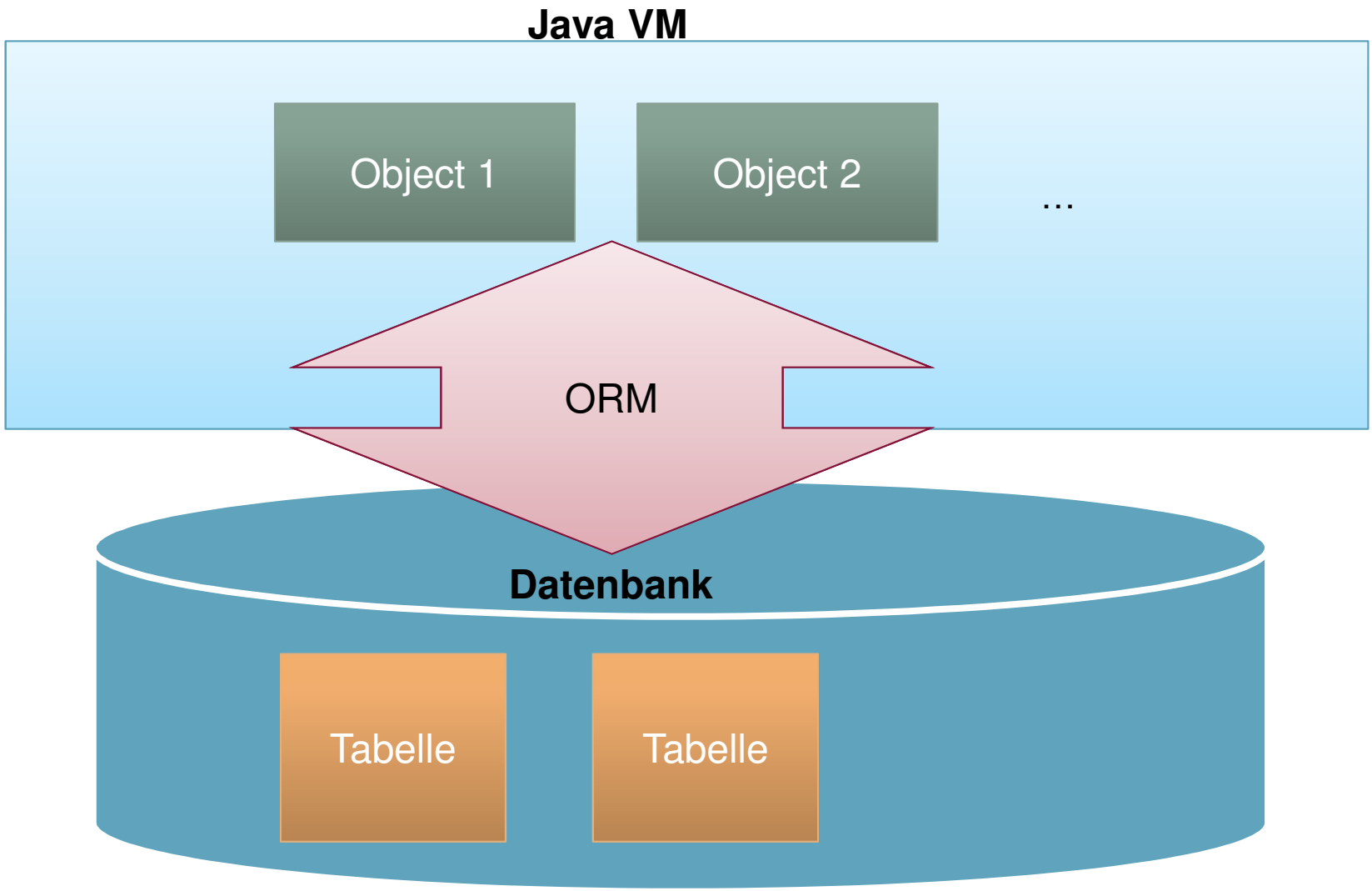
Java Persistence API

JEE 5/6 Schulung

.consulting .solutions .partnership

msg systems

Objektrelational Mapping



JPA JSR Überblick

Java Persistence API 2.0 JSR-317 – Dezember 2010

- Support for collections of embedded objects
- Ordered list
- Improvments in JPQL data bindings of lists
- Unidirectional OneToMany without Join Table
- Type safe Criteria API
- Support for Validation

Java Persistence API JSR 220 - Mai 2006

- JPA 1.0 ist Bestandteil der EJB 3.0 Spezifikation und gehört zur Java EE 5 Spezifikation
- JPA ersetzt CMP (Container-Managed Persistence) der J2EE Spezifikation
- JPA übernimmt viele Konzepte von Hibernate
- JPA 1.0 spezifiziert eine Teilmenge der Hibernate-Funktionsumfänge
- JPA kann innerhalb und außerhalb des EJB-Containers verwendet werden
- …

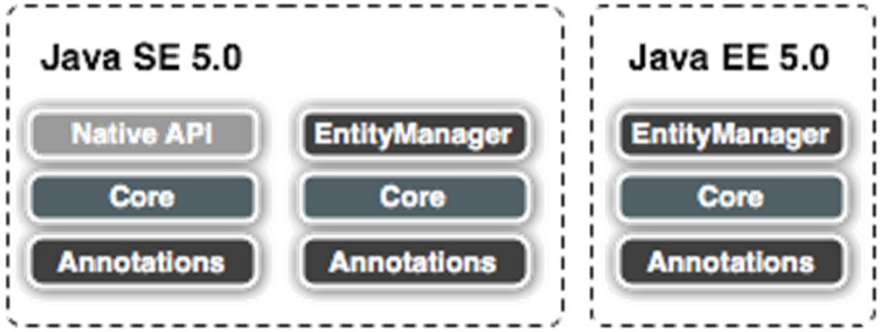
Wichtige JPA Implementierungen

JPA Implemtierungen

- JBOSS Hibernate
- EclipseLink (früher Oracle Toplink)
- OpenJPA
- DataNucleus
- ObjectDB
- IBM im Websphere Application Server

Hibernate kann in 2 Varianten eingesetzt werden

- Native Hibernate API
- JPA konforme Implementierung auf Java SE oder Java EE



In diesem Workshop verwenden wir JPA API.

Die native Hibernate API und Hibernate-spezifische Erweiterungen zur JPA API werden nicht betrachtet.

Warum JPA ?

Persistente Objekte sind POJO‘s

- Trennung von Business Logik und technischem Persistenz Framework
- Eine Entität kann von unterschiedlichen Frameworks gemappt werden (z. B. Persistenz in DB und XML)
- Portierbarkeit
- Testbarkeit

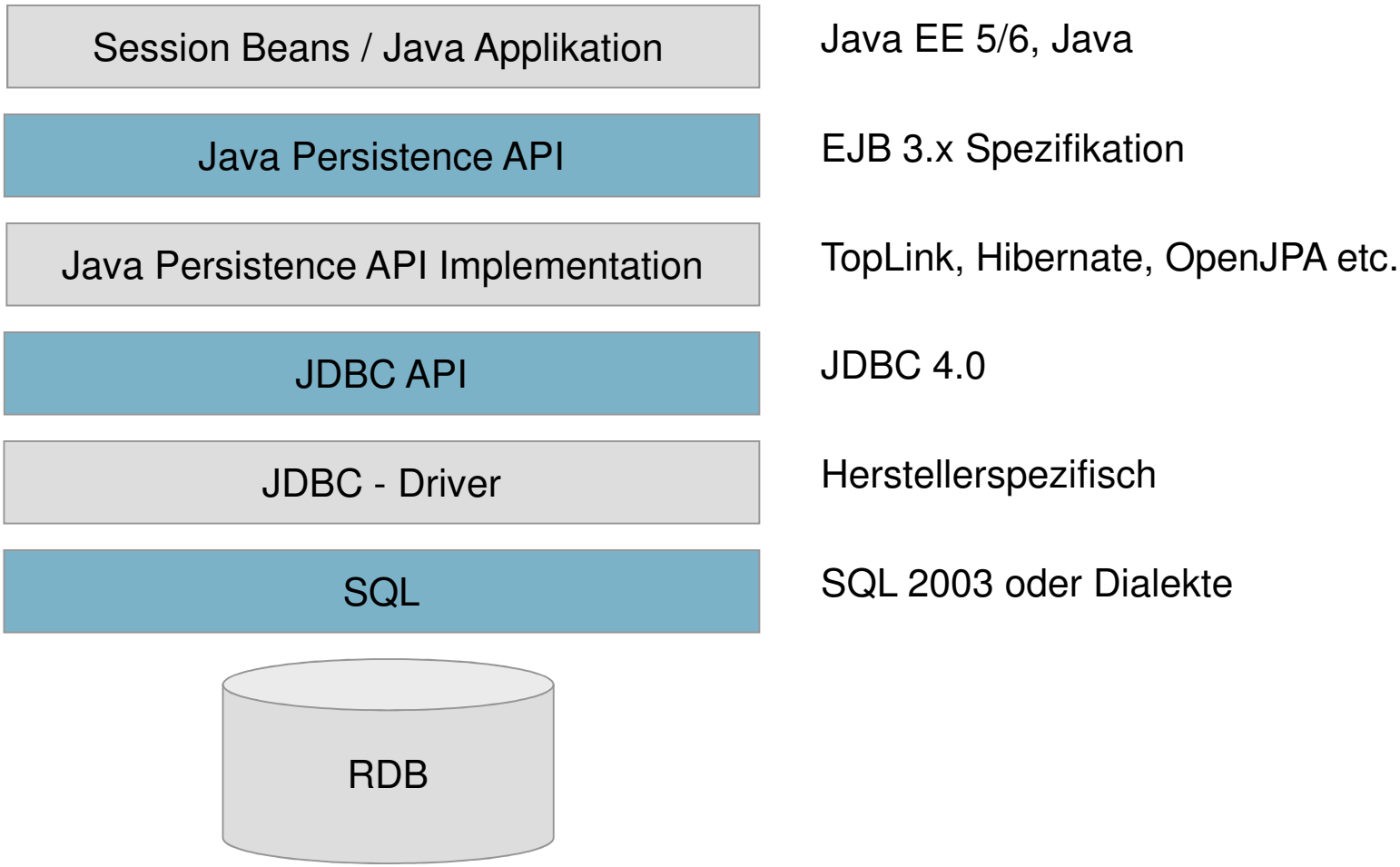
Transparentes Lifecyclemanagement (Tx basiert)

- Business Logik muss keine zusätzlichen Lifecycle Methoden implementieren (kein save(), oder update())
- Wenig Overhead
- Transaktionssteuerung lässt sich über den Container managed
- Testbarkeit

Convention over Configuration

- Verringert Overhead in der Konfiguration

Schichten


Session Beans / Java Applikation
Java EE 5/6, Java
Java Persistence API
EJB 3.x Spezifikation
Java Persistence API Implementation
TopLink, Hibernate, OpenJPA etc.
JDBC API
JDBC 4.0
JDBC - Driver
Herstellerspezifisch
SQL
SQL 2003 oder Dialekte
RDB

JPA Entity, Entity Manager

JPA Entity

```
public class Customer {


        private Long id;


        private String vorname;


        private String name;
}
```

JPA Entity

```
@Entity
public class Customer {

        @Id
        @GeneratedValue
        private Long id;

        @Column
        private String vorname;

        @Column
        private String name;
}
```

JPA Annotationen I

@Entity

Definiert ein Objekt als Entity

- Objekt kann damit als Entity in JPA verwendet werden
- Über optionales Element `name` kann der Entity Name gesetzt werden

```
@Entity
public class MyEntity{
```

@Column

Definiert das Attribut oder die Property als Tabellen Feld

- optional: muss nicht angegeben werden
- Kann auf Fields, oder getter angewendet werden
- Mittels name kann der Name der Tabellenspalte spezifiziert werden

```
@Column
public String getDescription{…}
```

@Id

Definiert die Eigenschaft als ID (Primary Key) der Entität

```
@Id
@Column
public Long oid;
```

JPA Entity

```
@Entity
@Table(name="CUSTOMER_TABLE")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "FIRST_NAME")
    private String vorname;

    @Column
    private String name;
}
```

@Generated Value

Definiert ein ID Feld als automatisch generiert. Die Art der Generierung kann dabei angegeben werden

- Optionale Parameter. Wird nichts angegeben, wählt das Framework selber die Strategie
- Parameter
  Generator : der Name des Generators
  Strategy: Auto, Identity, Sequence oder Table

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "S_TRANSPORTROUTE")
@SequenceGenerator(name = "S_TRANSPORTROUTE„,
                   sequenceName = "S_TRANSPORTROUTE",
                   allocationSize = 1)
public Long oid;
```

JPA Entitäten: POJOs

- **P**lain **o**ld **J**ava **O**bjects
- Persistente Klassen sind POJOs
- Persistente Klasse müssen nicht von einer Klasse abgeleitet werden. (Wie z.B. bei Entity Beans)
- Persistente Klassen können auch außerhalb des Persistenzkontextes wie normale Javaklassen verwendet werden.
- Persistente Klassen wissen nichts von der darunterliegenden Datenbank.

JPA Entitäten: Eigenschaften

- Klasse darf kein Interface und keine Enumeration sein
- Jede Entity muss einen Primärschlüssel (@Id) haben.
- Jede Entity muss einen leeren Defaultkonstruktor besitzen

Configuration by Exception - Conventions ov

- JPA verfolgt das Prinzip „Configuration by Exception“
  - Konfiguration der Ausnahmen
  - Voreinstellungen (Conventions) wenn keine Konfiguration z.B. in Form einer Annotation angegeben wird.
  - Man muss die Conventions kennen, um zu verstehen wie JPA funktioniert.

Entity Manager – außerhalb EJB Container

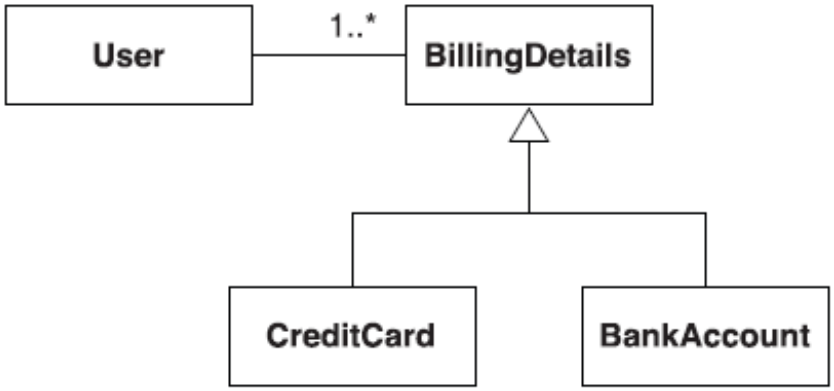
```
EntityManagerFactory emf;
EntityManager em;

emf = Persistence.createEntityManagerFactory("acm");
em = emf.createEntityManager();

em.getTransaction().begin();
Customer customer = new Customer();
customer.setName("Meier");
customer.setVorname("Peter");
em.persist(customer);
em.getTransaction().commit();
Customer cu = em.find(Customer.class, new Long(1));
```

persistence.xml

```
<persistence version="1.0" 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_1_0.xsd">
   <persistence-unit name=„test" transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
            <class>de.msg.jpa.entity.Customer</class>
            <properties>
                <!-- HSQLDB Datenbank auf localhost:-->
                <property name="hibernate.connection.username"
value="sa"/>
                <property name="hibernate.connection.password"
value=""/>
                <property name="hibernate.connection.url"
                          value="jdbc:hsqldb:hsql://localhost"/>
                <property name="hibernate.dialect„
                          value="org.hibernate.dialect.HSQLDialect"/>
                <property name="hibernate.show_sql" value="true"/>
                <property name="hibernate.format_sql" value="true"/>
                <property name="hibernate.debug" value="true"/>
                <property name="hibernate.hbm2ddl.auto" value="create"
/>
            </properties>
         </persistence-unit>
</persistence>
```

Relational vs. objektorientiert

Relational vs. objektorientiert: Vererbung



Relational vs. objektorientiert: Identität

- Identität in Java
  - Objekt Identität (identisches Objekt an der selben Stelle im Hauptspeicher). Wird mit `a == b` überprüft.
  - Inhaltlich gleich, wird mit `equals()`–Methode überprüft
- Identität in SQL
  - Identischer Primary Key

Identität

Technischer primary Key

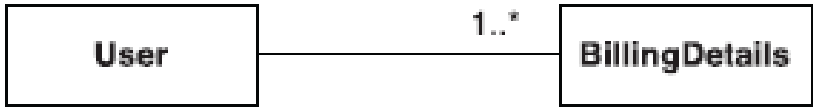
```
create table USERS (
    USER_ID bigint not null primary key,
    USERNAME varchar(15) not null unique,
    NAME varchar(50) not null,
...
)
create table BILLING_DETAILS (
    BILLING_DETAILS_ID bigint not null primary key,
    ACCOUNT_NUMBER VARCHAR(10) not null unique,
    ACCOUNT_NAME VARCHAR(50) not null,
    ACCOUNT_TYPE VARCHAR(2) not null,
    USER_ID bigint foreign key references USER
)
```

Objekt-Assoziationen vs. Relationen

Java

Assoziationen werden über Objekt-Referenzen abgebildet.

```
public class User {
    private Set billingDetails;
    ...
}
public class BillingDetails {
    private User user;
    ...
}
```



Soll die Assoziation in beide Richtungen navigierbar sein, ist für jede Richtung eine Assoziation notwendig.

Objekt-Assoziationen vs. Relationen Datenbank

- Relationen über Fremdschlüssel haben keine „Richtung“.
- „Navigation“ gibt es bei relationalen Datenbanken nicht.
- Die Assoziationen werden durch Table Joins umgesetzt.
- Für many to many Relationen ist eine Zwischentabelle (join table, link table, cross table) notwendig.

```
create table USER_BILLING_DETAILS (
    USER_ID bigint foreign key references USERS,
    BILLING_DETAILS_ID bigint foreign key references
    BILLING_DETAILS,
    PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)
)
```

Navigation im Objekt-Graphen

Beispiel

Zugriff auf Javaobjekt Kontonummer

```
aUser.getBillingDetails().getAccountNumber()
```

```
select *
from USERS u
left outer join BILLING_DETAILS bd on bd.USER_ID
   = u.USER_ID
where u.USER_ID = 123
```

Problem:

Es sind pro Knoten im Objekt-Graphen ein select-Statement notwenig. (*n+1 selects problem*)

Vorgehen

Objektorientiert -> relational

- Es wird objektorientiert modelliert.
- Die Datenbanktabellen werden generiert

Positiv: gutes Objektmodell

Negativ:

- schlechtes Datenbankdesign
- schlechte Performance

Relational -> Objektorientiert

- Datenbankschema ist vorgegeben
- Aus dem Datenbankschema werden die Javaobjekte generiert

Positiv: gutes ER-Modell

Negativ:

- schlechtes Objektmodell
- Objekte passen nicht darüberliegenden Schichten
- Navigation im Objektgraphen ist aufwändig.

Vorgehen

Getrennte Entwicklung des Objektmodells und des ER-Modells

- Objektmodell wird erstellt
- Unabhängig davon wird das ER-Modell erstellt

Positiv:

- gutes Objektmodell
- gutes ER-Modell

Negativ:

- komplexes OR-Mapping
- schlechte Performance

Gemeinsame Entwicklung des Objektmodells und des ER-Modells

- Objektmodell wird gemeinsam/parallel mit dem ER-Modell entwickelt

Positiv:

- einfaches OR-Mapping
- bessere Performance
- geringstes Übel

Negativ:

- Kompromisse im Objektmodell und im ER-Modell

JPA im EJB Container

Entity Manager Verwendung

Nicht im EJB Container

```
emf = Persistence.createEntityManagerFactory(„test");
em = emf.createEntityManager();
em.getTransaction().begin();
```

Innerhalb des EJB Containers (Container Managed)

```
@PersistenceContext
 EntityManager em;
```

Innerhalb des EJB Containers (Application Managed)

```
@PersistenceUnit
 EntityManagerFactory emf;
```

persistence.xml

Nicht im EJB Container

```
<persistence-unit name="acm" transaction-
    type="RESOURCE_LOCAL">

<property name="hibernate.connection.driver_class"
    value="org.hsqldb.jdbcDriver"/>
<property name="hibernate.connection.username"
    value="sa"/>
<property name="hibernate.connection.password" value=""/>
<property name="hibernate.connection.url"
    value="jdbc:hsqldb:hsql://localhost"/>
```

Innerhalb des EJB Containers

```
<persistence-unit name="acm">
<jta-data-source>
    java:/caveatemptorTestingDatasource</jta-data-source>
```

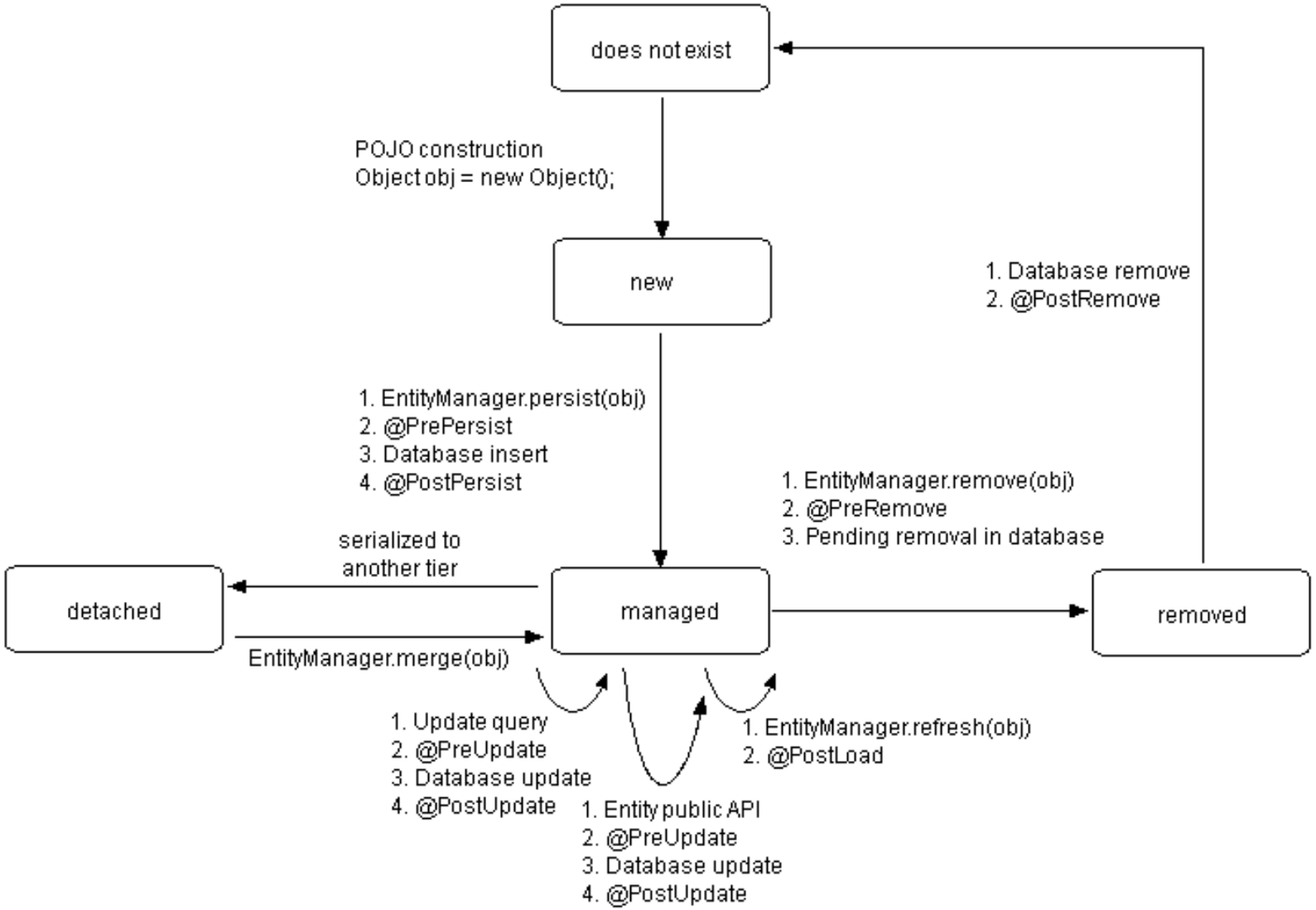
Zwei Lifecyclealternativen durch den Container

Transaction (Default)	**Der Persistenzkontext ist an die aktuelle Transaktion gebunden**
	`@PersistenceContext(`
	`  type=PersistenceContextType.TRANSACTION)`

Extended	**Ist an den Zustand eines Stateful Bean gebunden**
	▪ Lazy Loading ist auch außerhalb der Business Schicht möglich
	▪ Entitäten können über den Scope einer einzelnen Transaktion gespeichert und verwendet werden
	`@PersistenceContext(`
	`  type=PersistenceContextType.EXTENDED)`

Entity Lifecylce

JPA Entity Lifecycle Callback Event Annotations



Detached Entities

- Entities können explizit oder implizit detached werden
- detachete und geänderte Entitäten können mit merge wieder persisitiert werden.
- Löschen einer Entity aus dem Objektgraphen wird nicht erkannt.
- Operationen, die nicht von merge automatisch ausgeführt werden, müssen mitprotokolliert werden. (z.B. Löschen)
- Detachte Entities sind nicht transaktionssicher!
- Enities nicht direkt sondern über Interface zum Client geben.
- Vorsicht beim Detach von Objekten mit Lazy Fetching von Assoziationen

Assoziationen

JPA Objekt Referenzen

@OneToOne	**Definiert eine 1:1 Beziehung** - Methoden: Shared Primary Key, Join Column , Join Table - Default Foreign Primary Key auf der definierenden Seite - Sollte mit Foreign Key verwendet werden - Kann undirektional auf definierender Entität sein
@ManyToOne @OneToMany	**Definiert die jeweiligen Seiten einer 1:n Beziehung** - Methoden: Join Column, Join Table - Nur ManyToOne kann unidirektion definiert werden - Sollte mit Foreign Key verwendet werden - Bidirektion wird OneToMany über „mappedBy“ definiert
@ManyToMany	**Definiert eine n:m Beziehung** - Über Join Table - Bidirektion wird ManyToMany an der nicht definierenden Seite über „mappedBy“ definiert

`@OneToOne` (foreign key)

```
@Entity
public class Customer implements Serializable {
   @Id
   private Long id;
   @OneToOne
   @JoinColumn(name="passport_fk")
   private Passport passport;
   }

@Entity
public class Passport implements Serializable {
   @OneToOne(mappedBy = "passport")
   private Customer owner;

}
```

Mapping durch foreign key.
Spalte der Tabelle Customer mit dem FK auf Tabelle Passport

„passport“ ist der Attributname in der Klasse Customer auf den die Entity Passport gemappt wird.

- Ohne `@OneToOne(mappedBy… )` : unidirektional.

@ManyToOne (join column)

```
@Entity
public class Flight implements Serializable {
   ...
   @ManyToOne
   @JoinColumn(name="COMP_ID")
   private Company company;
   ...
}
```

`@OneToMany` (bidirectional, ManyToOne as owning side)

```
@Entity
public class Company {
   @OneToMany(mappedBy="company")
   Set<Flight> flights;
   ...
}

@Entity
public class Flight {
   @ManyToOne
   @JoinColumn(name="company_fk")
   Company company;
   ...
}
```

`@OneToMany` (unidirectional, JoinColumn)

```
@Entity
public class Customer implements Serializable {
   @OneToMany
   @JoinColumn(name="CUST_ID")
   Set<Ticket> tickets;
...
}

@Entity
public class Ticket implements Serializable {
... //no bidir
}
```

Assoziationen zwischen Entitäten

`@ManyToMany` (JoinTable)

```
@Entity
public class Employee {
   @Id
    private int id;

   private String name;

   @ManyToMany
   @JoinTable(name="EMP_PROJ",
      joinColumns=@JoinColumn(name="EMP_ID"),
      inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
   private Collection<Project> projects;
   //...
}

@Entity
public class Project {
   @Id
    private int id;

    private String name;

    @ManyToMany(mappedBy="projects")
   private Collection<Emploee> employees;
   //...
}
```

Eager/Lazy fetching

Lazy/Eager Loading

Lazy

Beziehungen werden erst beim 1. Zugriff geladen

- Beim Serializieren oder Detachen sind die Relationen ggf. nicht gefüllt und können nicht mehr gefüllt werden
- Performance besser, wenn nur selten auf Beziehungen zugegriffen wird

Eager

Beziehungen werden beim Laden der Parent Entität sofort geladen

- Kein Problem beim Detachen oder Serializieren
- Ggf. werden zuviele Daten gelesen und/oder zuviele Objekte im Speicher erstellt

Lazy/Eager Loading ist auf unterschiedlichen Granularitäten verfügbar

- Lazy (single-point) associations (OneToOne oder ManyToOne)
- Lazy Collections (OneToMany oder ManyToMany)
- Lazy Properties – Attribute eines persistenten Objektes

lazy (single-point) associations

```
@Entity
@Table(name = "ADDRESS")
public class AddressEntity implements Serializable {

    ...

    @OneToOne(fetch = FetchType.LAZY)
    @PrimaryKeyJoinColumn
    private User user;
```

lazy collections

Referenzen einer Collection auf andere persistente Objekte bei one-to-many und many-to-many

```
@Entity
@Table(name = "ITEM")
public class Item {
   ...

   @OneToMany(cascade = CascadeType.ALL, fetch =
            FetchType.LAZY)
   @JoinColumn(name = "ITEM_ID", nullable = false)
   private List<Bid> bids = new ArrayList<Bid>();
```

lazy properties

Auch Attribute eines persistenten Objektes können lazy geladen werden. (Disabled by default)
Macht nur Sinn bei Tabellen mit sehr vielen Spalten und sehr vielen Daten in den einzelnen Spalten.

```
@Basic(fetch = FetchType.LAZY)
private BigDecimal initialPrice;
```

Lazy-Loading in Hibernate über Proxy-Klassen

- Beim Laden einer Entity erzeugt Hibernate für jedes referenzierte Entity zunächst eine Instanz einer Proxy-Klasse.
- Dieses Proxy-Objekt kennt den PK des Entities.
- Beim Aufruf von getter oder setter-Methoden, lädt das Proxy-Objekt das eigentliche Entity.
- Ab jetzt werden alle Aufrufe direkt an das Entity weitergeleitet.
- Das Austauschen der Entities gegen die Proxy Objekte erfolgt zur Laufzeit per Bytecodemanipulation.
- Andere JPA-Provider ergänzen den Byte-Code nach dem Compilieren um die Persistenz-Funktionalität. (z.B. Enhancer bei OpenJPA / Kodo)

EAGER/Lazy-Fetching

- Default bei one-to-one und many-to-one
  - FetchType.EAGER
- Default bei one-to-many und many-to-many
  - FetchType.LAZY
- Defaultverhalten kann übersteuert werden.
  z.B.
  - @OneToMany(fetch = FetchType.EAGER)

Vorsicht bei OneToMany und FetchTyper.EAGER!

Bei großen Resultsets werden sehr viele Objekte erzeugt!

Cascading

Transitive Persistenz (Cascading)

- Transitive Persistenz erlaubt die Weitergabe von JPA-Operationen, wie beispielsweise `merge(), remove()` und `persist()` auf in Beziehung stehende POJOs.
- Angenommen ein POJO A hat eine 1:n Assoziation zu POJO B.
- Falls auf A merge() aufgerufen wird, erfolgen standardmäßig keine Aufrufe von merge() auf B, die an A hängen.

Transitive Persistenz (Cascading)

- Für die Methoden merge(), persist(), refresh(), remove des EntityManagers gibt es einen entsprechenden CascadeType:
- `MERGE, PERSIST, REFRESH, REMOVE, ALL`

Wird eine Operation auf einem POJO ausgeführt, so wird die Operation auch auf den POJOs ausgeführt, die über Assoziationen verknüpft sind.

→ Cascade wirkt auch auf Collections.

- ab JPA2 mittels DETACH kann die Existenz eines Objektes an die Existenz innerhalb einer bestimmten Collection gebunden werden
- Übernommen aus Hibernate:

```
@OneToMany(orphanRemoval=true)
public Set<Order> getOrders() { return orders; }
```

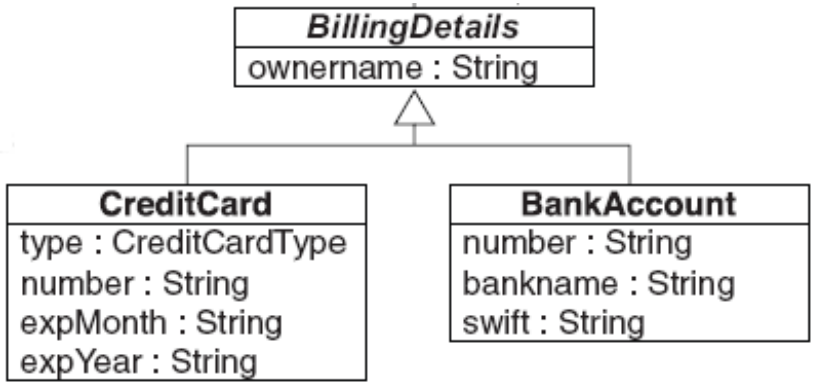
Transitive Persistenz (Cascading)

Beispiel:

```
@Entity
@Table(name = "ITEM")
public class Item {
    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id = null;

    @OneToMany(cascade = CascadeType.ALL, fetch =
        FetchType.LAZY)
    @JoinColumn(name = "ITEM_ID", nullable = false)
    private List<Bid> bids = new ArrayList<Bid>();
```

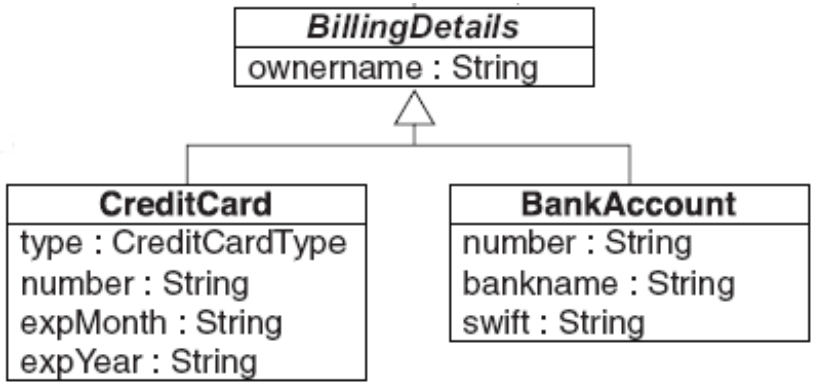
Vererbung

Vererbung

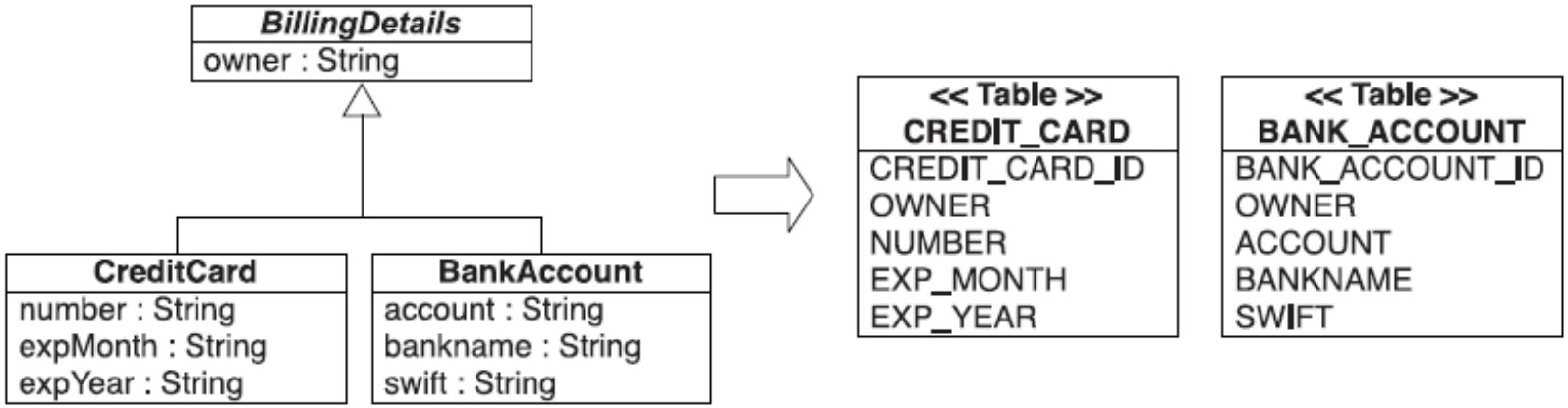

BillingDetails
ownername : String
CreditCard
type : CreditCardType
number : String
expMonth : String
expYear : String
BankAccount
number : String
bankname : String
swift : String

Vererbung

- Table per concrete Class
- Single Table per class hierarchy
- Table per subclass



Vererbung: Table per concrete Class



- Jede konkrete Klasse wird auf eine Tabelle abgebildet.
- Assoziationen auf die Superklasse sind problematisch, weil Relationen über Foreign Key auf Tabellen nicht möglich sind. ManyToOne zwischen BillingDetail und User wäre nicht möglich.
- Queries auf die Superklasse müssen durch mehrere SQLs auf alle Tabellen umgesetzt werden.

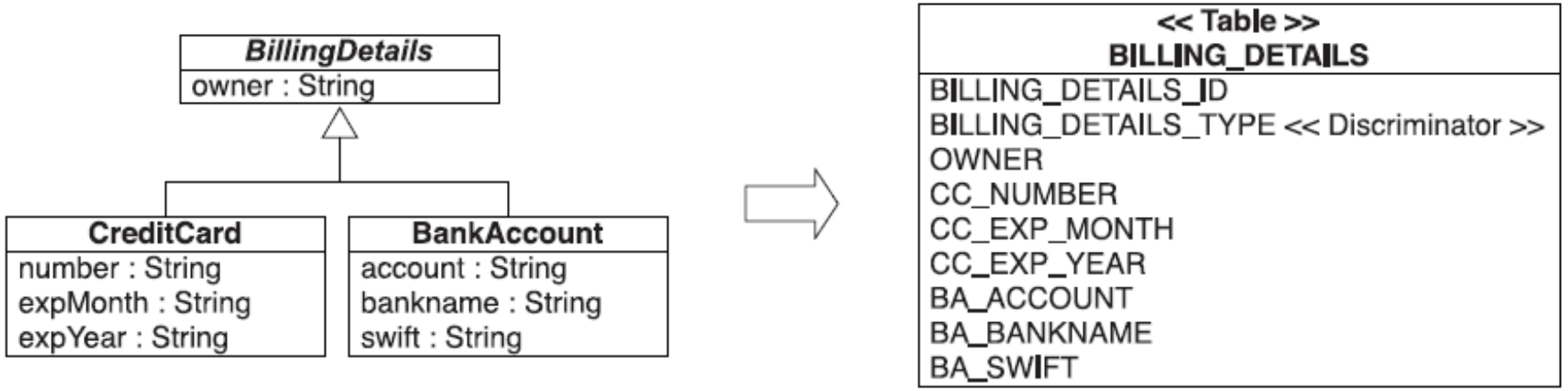
Vererbung: Table per concrete Class

```
@MappedSuperclass
public abstract class BillingDetails {
   @Column(name = "OWNER", nullable = false)
   private String owner;
        ...
}
```

SubClass:

```
@Entity
public class CreditCard extends BillingDetails {
   @Id @GeneratedValue
   @Column(name = "CREDIT_CARD_ID")
   private Long id = null;
   @Column(name = "NUMBER", nullable = false)
   private String number;
   ...
}
```

Vererbung: Table per class hierachy



- Die gesamte Klassenhierarchie wird auf eine Tabelle gemappt.
- Die Diskriminator-Spalte legt fest, von welchem Typ die Subklasse ist.
- Gute einfache, performante Lösung
- Problem: Attribute einer Subklasse müssen als nullable deklariert werden.

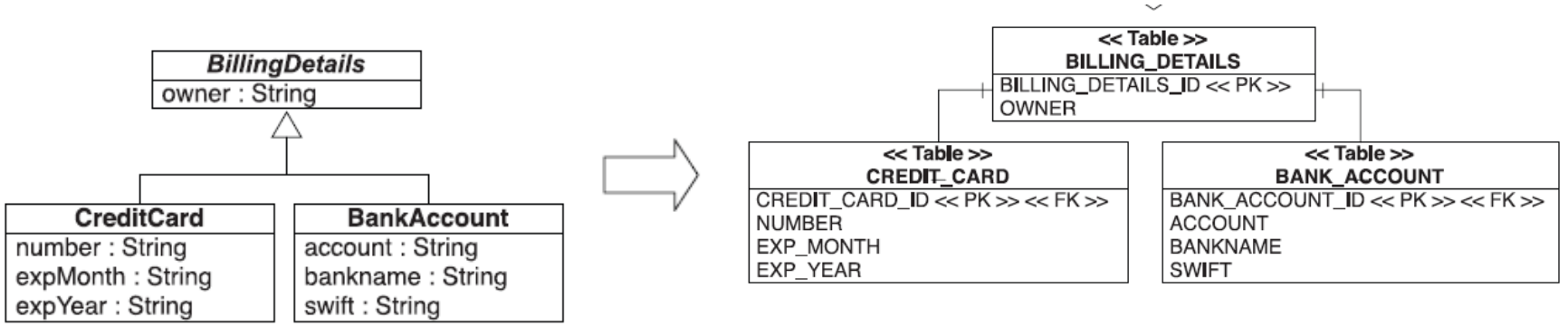
JPA Mapping Table per class hierachy

```
@Entity
@Inheritance(strategy =
   InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
   name = "BILLING_DETAILS_TYPE",
   discriminatorType = DiscriminatorType.STRING
)
public abstract class BillingDetails {
   @Id @GeneratedValue
   @Column(name = "BILLING_DETAILS_ID")
   private Long id = null;
   @Column(name = "OWNER", nullable = false)
   private String owner;
   ...
}
```

SubClass:

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
   @Column(name = "CC_NUMBER")
   private String number;
}
```

Vererbung: Table per subclass



- Attribute der Superklasse werden in einer Tabelle abgelegt.
- Für jeder Subklasse gibt es eine eigene Tabelle mit den zusätzlichen Attributen der Subklasse
- Problem: Join ist notwendig. (Joined Table Inheritance)

JPA Mapping Table per subclass

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
   @Id @GeneratedValue
   @Column(name = "BILLING_DETAILS_ID")
   private Long id = null;
   ...
}

@Entity
public class BankAccount extends BillingDetails{
   ...
}

@Entity
public class CreditCard extends BillingDetails{
   ...
}
```

Queries

Flushing

JPA *EntityManager* implementiert *transparent write behind.*

- Änderungen am Domainmodell werden nicht sofort in die Datenbank geschrieben.
- JPA sammelt die Änderungen, um die Datenbankzugriffe zu minimieren.

JPA flush wird ausgeführt, wenn

- Die Transaktion commitet wird
- Die Applikation explizit `EntityManager.flush()` aufruft.

JPA ruft nicht vor jeder Query flush auf.

- Wenn es aber Änderungen im Speicher gibt, die das Ergebnis der Query beeinflussen, synchronisiert JPA zuerst.
- Dies entspricht der Default-Einstellung `FlushMode.AUTO`
- `FlushMode.COMMIT` Flush nur bei commit
- `FlushMode.Never` Flush muss in der Applikation explizit aufgerufen werden

```
EntityManager.setFlushMode()
```

Entities laden

- In JPA gibt es verschiedene Optionen Entitäten zu laden:
  - Eine einzelne Instanz über die ID laden
  - Navigation auf dem Objektgraphen
  - Queries in der Java Persistence Query Language (JPQL)
  - Queries in SQL (NativeQuery)

- In JPA 2.0 kommt neu die Criteria API hinzu

Java Persistence Query Language (JPQL)

- JPQL ist eine mächtige Abfragesprache basierend auf dem Entitätenmodell:
  - Stark an SQL angelehnt
  - Unabhängig von der darunterliegenden Datenbank
  - Abfragen basieren auf dem Klassenmodell (Entitäten), nicht auf dem Datenmodell (Tabellen)
  - Unterstützt OO-Konstrukte wie Vererbung, Polymorphismus und Pfadausdrücke

Beispiel:

```
String queryString =
   "select e.address from Employee e where
   e.mainProject.name = `JPA Kurs`";
Query query = em.createQuery(queryString);
List<Address> adresses = query.getResultList();
```

JPQL Features

JPQL ist eine ausdrucksstarke und flexible Abfragesprache.

- JOINS und Subqueries (`IN, JOIN, EXISTS`)
- Aggregatsfunktionen (`AVG, COUNT, MIN, MAX, SUM`)
- `GROUP BY` und `HAVING`
- Funktionen (`LOWER, ABS, TRIM` ...)
- `LIKE`
- Collection-Abfragen: `IS EMPTY, MEMBER`
- `ANY, ALL, SOME`

Dynamische Queries

- Bei Dynamischen Queries wird der JPQL String zur Laufzeit erstellt.
  - Kontextabhängige Queries
  - String Concatenation

Beispiel:

```
EntityManager em = ...
String queryString =
   "select e from Employee e where e.address.city
   = 'Bern'";
Query query = em.createQuery(queryString);
List<Employee> employees =
   query.getResultList();
```

Named Queries

- Named Queries werden statisch definiert und können überall in der Applikation verwendet werden.
- Die JPA Infrastruktur kann Named Queries vor der eigentlichen Ausführung parsen und kompilieren (Prepared Statements)
  - Parsen/Kompilierung muss nur einmal durchgeführt werden
  - Kann beim Deployen/Startup erfolgen und überprüft werden (Fail Fast)

Beispiel:

```
@NamedQuery(name = "Employee.findAll",
                query = "SELECT e FROM Employee e")
public class Employee { ... }
```

```
EntityManager em = ...
Query q = em.createNamedQuery("Employee.findAll");
List<Employee> employees = query.getResultList();
```

JPQL: Named Parameter

```
String queryString = “select item from Item item“
                   + “where item.description like
   :searchString “;
List result = em.createQuery(queryString)
                   .setParameter(“searchString“,
   searchString)
                   .getResultList();
```

JPQL: Positional Parameter

```
String queryString = “from Item item“
                   + “where item.description like ? “;

List result = em.createQuery(queryString)
                    .setParameter(0, searchString)
                    .getResultList();
```

Probleme

- Schlechter lesbar
- Parameter können leichter vertauscht werden
- Der erste Parameter wird mit 0 angegeben. Es kann leicht fälschlicherweise der Parameter 1 statt 0 angegeben werden.
- Bei Änderungen an der Query kann die Reihenfolge verändert werden und Parameter dadurch falsch gesetzt werden.

Empfehlung: Besser named parameter verwenden.

Pfad-Ausdrücke

Ein Pfadausdruck ermöglicht die direkte Navigation von einem äusseren zu inneren, referenzierten Objekten:

```
SELECT e.address      FROM Employee e
SELECT e.address.name FROM Employee e
```

Ein Pfadausdruck kann in einer *Collection* enden:

```
SELECT e.projects FROM Employee e
```

JPA Paging

Beispiel:

```
Query query = em.createQuery(“select u from User u order by u.name asc");
query.setFirstResult(0);
query.setMaxResults(10);
List<User> results = query.getResultList();
```

JPQL: Logische Operatoren

Beispiel:

```
from User u where u.firstname like 'G%' and u.lastname
   like 'K%'
```

Alle User deren Vorname mit G und deren Nachname mit K anfängt.

```
from User u where (u.firstname like 'G%' and
   u.lastname like 'K%') or u.email in
   ("foo@hibernate.org", "bar@hibernate.org")
```

Alle User deren Vorname mit G und deren Nachname mit K anfängt oder deren Email-Adresse `"foo@hibernate.org"` oder `"bar@hibernate.org"` ist.

JPQL: Sortierung

Beispiele:

```
from User u order by u.username
```

Alle User sortiert nach Username.

```
from User u order by u.username desc
```

Alle User absteigend sortiert nach Username.

```
from User u order by u.username asc, u.firstname asc
```

Alle User sortiert nach Username und Vorname.

Subselects

- Ein Subselect ist eine select Query, die in eine andere Query eingebettet ist.
- JPQL unterstützt Subselects in der `WHERE` Klausel.

Beispiel

```
Query query=em.createQuery("SELECT emp FROM
Employee emp WHERE emp.empSalary >
(SELECT avg(emp2.
empSalary) FROM Employee emp2)");
```

Vorteile und Nachteile von JPQL

- Vorteile
  - Sehr mächtig und flexibel
  - Stark an SQL angelehnt
- Nachteile
  - JPQL ist eine embedded Language die in Java mittels Strings verwendet wird.
    - Keine Überprüfung beim Kompilieren, keine Typ-Sicherheit
  - Flexible Komposition eines Queries ist nicht elegant möglich (String-Manipulation)
  - Für nicht-triviale Anwendungen ist SQL Knowhow und Verständnis des Datenmodels notwendig

SQL Queries

JPA ermöglicht die Formulierung von SQL-Queries:

```
Query q = em.createNativeQuery("SELECT * FROM emp
   WHERE id = ?",          Employee.class);
q.setParameter(1, employeeId);
List<Employee> employees = q.getResultList();
```

SQL-Queries können auch als NamedQuery definiert werden:

```
@NamedNativeQuery(
     name = "employeeReporting",
     query = "SELECT * FROM emp WHERE id = ?",
     resultClass = Employee.class)
```

Update und Delete Statements:

```
Query q = em.createNativeQuery("UPDATE emp SET salary
   = salary + 1");
int updated = q.executeUpdate();
```

Stored Procedures Aufruf:

```
entityManager.createNativeQuery("CALL STORED_PROC
   (....)");
```

Vielen Dank für Ihre Aufmerksamkeit

Marc Mährländer

XT – IT Architekt

Telefon: +49 89 96101-2482
vorname.name@msg-systems.com

www.msg-systems.com

.consulting .solutions .partnership