HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Datenbanksysteme 1
OR Mapping (Fortsetzung)

Prof. Stefan Keller

Dank an Prof. L. Bläser

Das Programm heute

- Lernkontrolle und ggf. Repetition „OR Mapping“
- „OR Mapping“ mit JPA - Fortsetzung
  - Wie kommen Entities und Daten zusammen?
  - JPA-Architektur (Begriffe)
  - Vererbung
  - Abfragen in JPA: JPQL
  - Kontrolle über Verhalten und Zustände der Entities (inkl. Aggregation und Komposition)
  - Selbsttstudium: Entity-Identität, Änderungen an Relations, Transaktionen
- Organisatorisches: Prüfung etc.

Lernziele heute

- Sie kennen OR Mapping-Varianten und wissen, wie man Entity-Identitäten und Vererbung in JPA abbildet.
- Sie haben einen Überblick über JPQL.
- Sie sind in der Lage, den JPA auch für komplexe OR-Mappings einzusetzen.

Zusammenfassung des Stoffs zu ORM bisher (Repetition)

- "Object-relational mapping" (ORM) ist ein Prozess...
- Java Persistence API (JPA): ein möglicher (relationaler) Ansatz dazu
  - Mittels JPA kann ein SW-Entwickler Daten von Java zu einer RDBMS mappen, speichern, verändern und abfragen
  - Kann in Java-EE und in Java-SE verwendet werden
  - Spezifikation mit mehreren Implementationen (sog. 'Pers. Provider')
  - Bekannt sind u.a. Hibernate, OpenJPA, DataNucleus und EclipseLink (= JPA-Referenzimplementation)
- Verbindung zur DB: persistence.xml
- Mapping zwischen POJOs und DB-Tabellen über Metadaten
  - JPA verwendet dazu Annotationen oder XML (orm.xml) oder beides
  - Das XML „übersteuert“ Annotationen
- Der SW-Entwickler arbeitet mit Objekten/Entitäten, nicht mit SQL

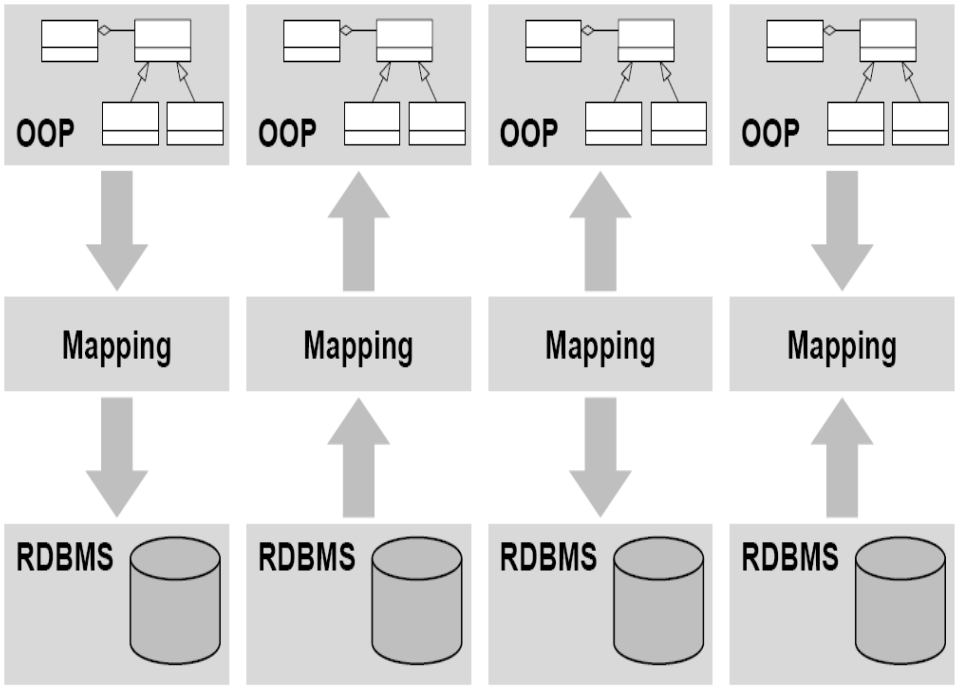
Wie kommen Entities und Daten zusammen?

- Top down (Forward Engineering).
  - Erstelle Business-Modell und erzeuge DB-Schema
- Bottom up
  - DB-Schema existiert; erzeuge daraus Business-Modell…
- Middle out
  - Erstelle Metadaten und generiere Java und DB-Schema…
- Meet in the middle
  - Business-Modell und DB-Schema existieren bereits: Erstelle Metadaten…
- Meta Model / Model driven
  - (hier für uns nicht so wichtig)

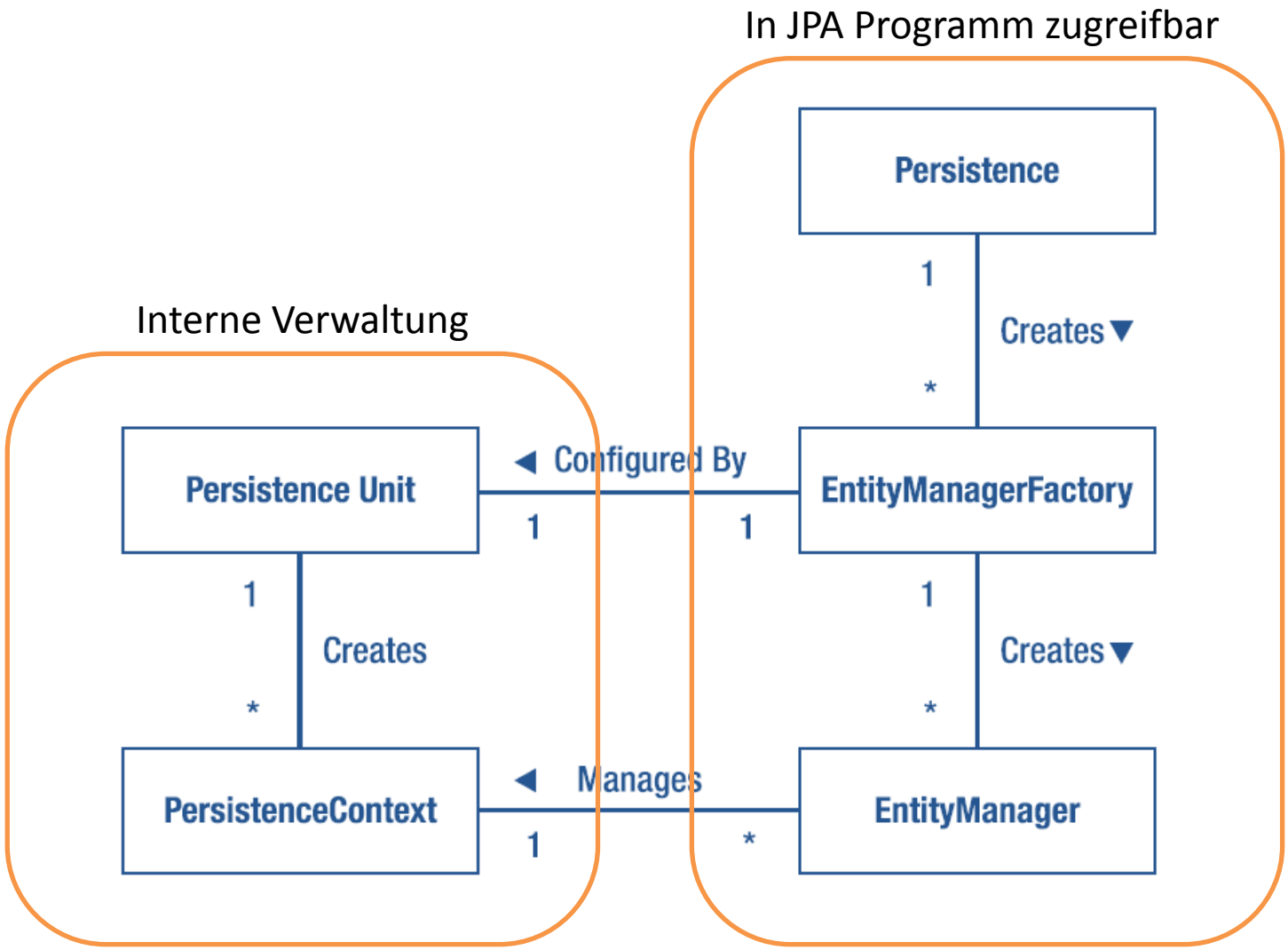
Welche Variante haben wir bisher immer verwendet?

OR-Mapping-Varianten

1. Forward Engineering (Code first, ‚Top-Down‘)
2. Bottom Up / Reverse Engineering (DB first)
3. Middle-out/Inside-out (Mapping first)
4. Meet-in-the-middle / Outside-in
5. Meta Model (‚model-driven‘)



JPA Architektur / Begriffe



JPA Persistence Unit

- Persistence Unit
  - Menge von Entity Klassen und deren Mapping
  - Bestimmt JPA Provider und DB-Anbindung
  - Durch META-INF/persistence.xml beschrieben
- EntitiyManagerFactory
  - Verwaltet eine Persistence Unit

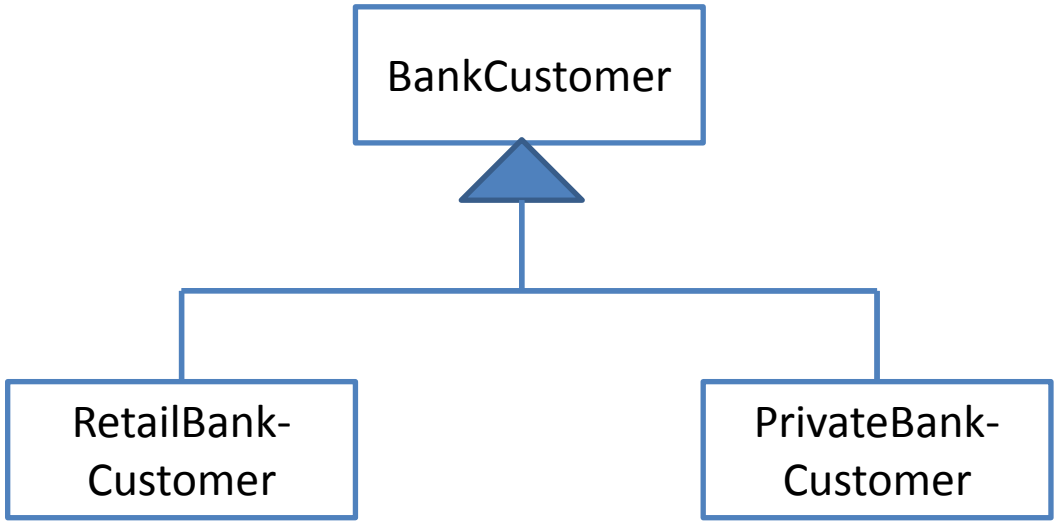
```
EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("Bank");
EntityManager em = factory.createEntityManager();
…
```

JPA Persistence Context

- Persistence Context
  - Verwaltet Menge von Entity Instanzen zur Laufzeit
  - Entity Instanz ist managed => in Persistence Context
  - Entity new oder detached => gehört zu keinem Context
  - Definiert transaktionelle Session
- Entity Manager
  - Verwaltet Persistence Context
  - Bietet Lifecycle-Operationen für Entity Instanzen an

Vererbung

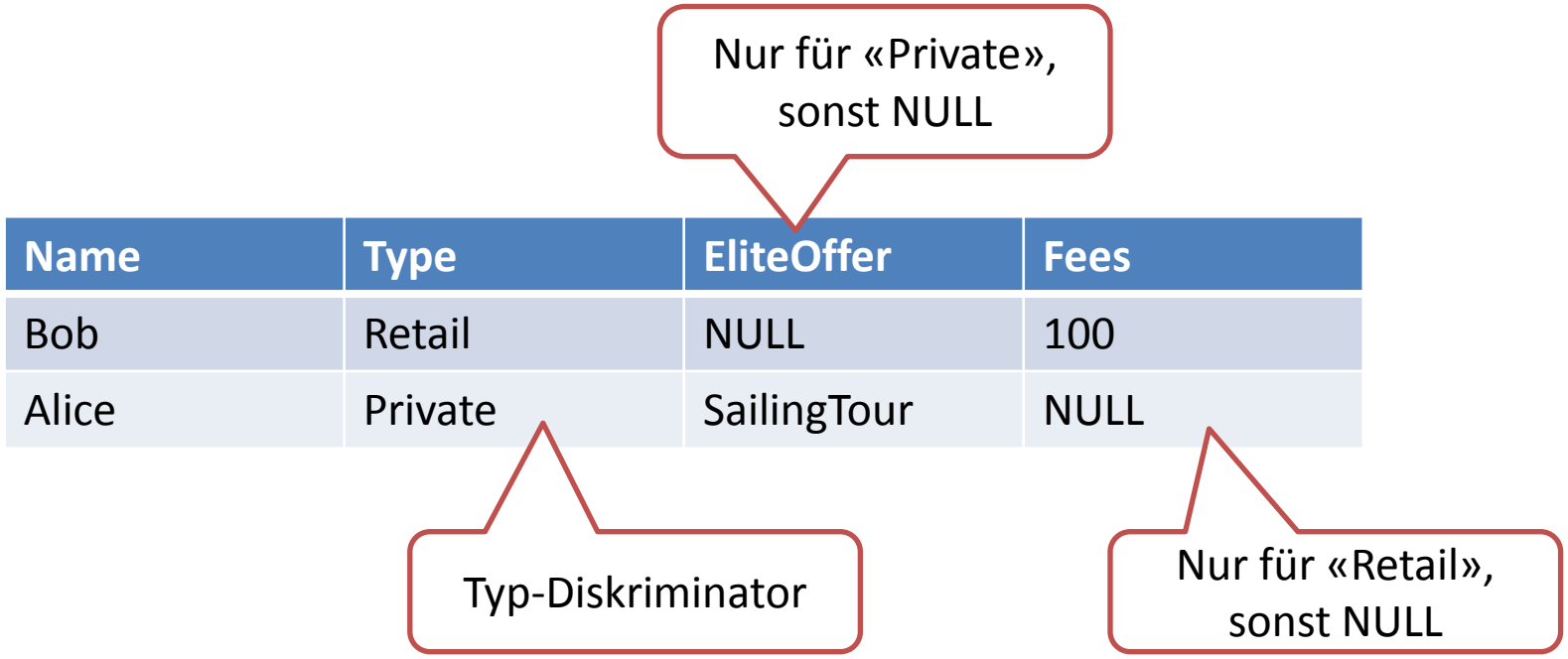
Vererbung: Abbildung



- Verschiedene Abbildungsstrategien
  - Single Table
  - Joined Table
  - Table Per Class

Kommen jemandem diese Strategien bekannt vor?

Single Table Mapping (1)



Name	Type	EliteOffer	Fees
Bob	Retail	NULL	100
Alice	Private	SailingTour	NULL

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type")
public abstract class BankCustomer {
  @Id private String name;
}
```

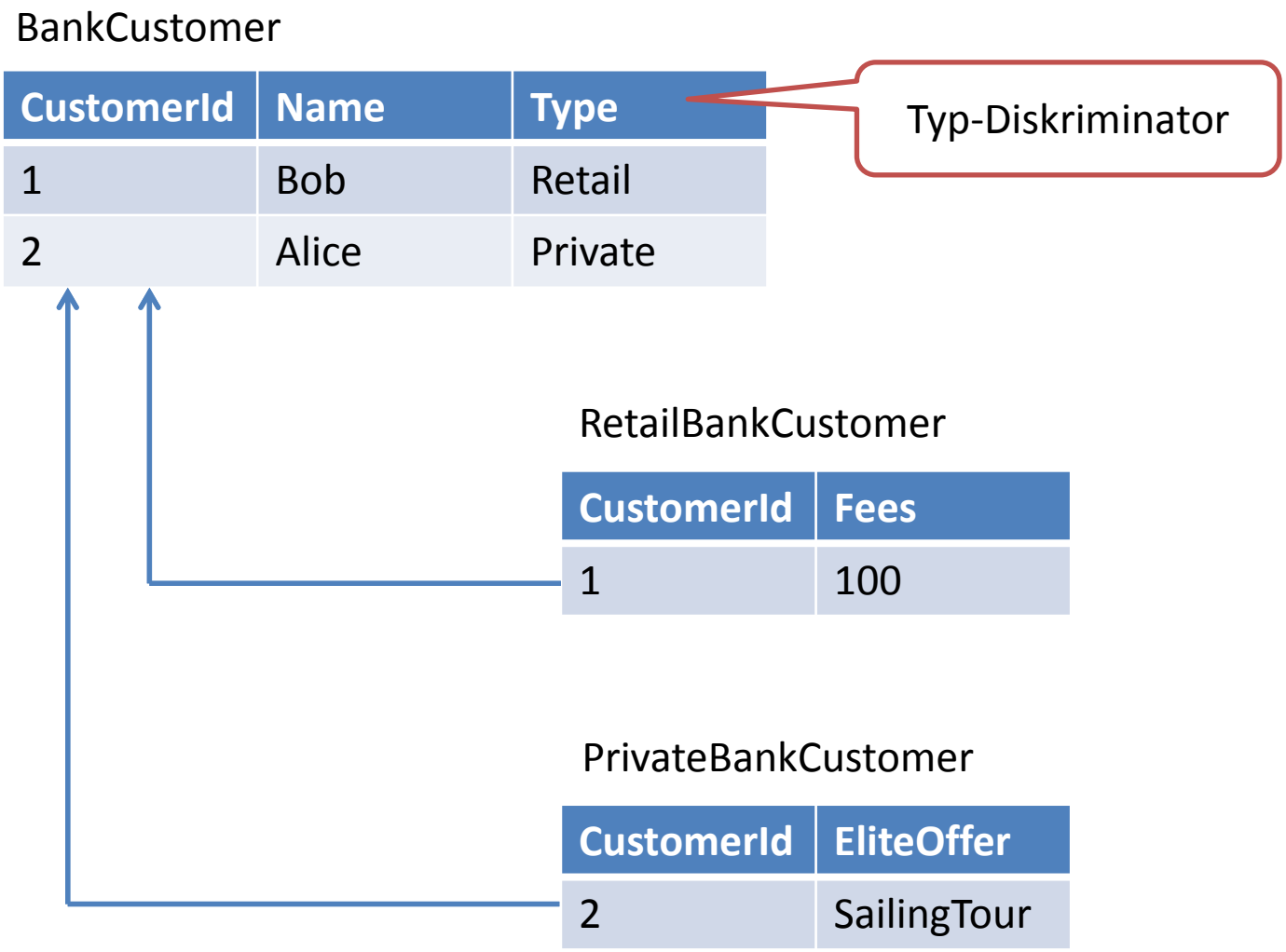
Single Table Mapping (2)

Name	Type	EliteOffer	Fees
Bob	Retail	NULL	100
Alice	Private	SailingTour	NULL

```
@Entity
@DiscriminatorValue("Retail")
public class RetailBankCustomer extends BankCustomer {
  private int fees;
}

@Entity
@DiscriminatorValue("Private")
public class PrivateBankCustomer extends BankCustomer {
  private String eliteOffer;
}
```

Joined Table Mapping (1)



Joined Table Mapping (2)

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "type")
public abstract class BankCustomer {
  @Id private int customerId;
  private String name;
}

@Entity
@DiscriminatorValue("Retail")
public class RetailBankCustomer extends BankCustomer {
  private int fees;
}

@Entity
@DiscriminatorValue("Private")
public class PrivateBankCustomer extends BankCustomer {
  private String eliteOffer;
}
```

Table Per Class Mapping

RetailBankCustomer

CustomerId	Name	Fees
1	Bob	100

PrivateBankCustomer

CustomerId	Name	EliteOffer
2	Alice	SailingTour

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BankCustomer {
  @Id private int customerId;
  private String name;
}
```

```
@Entity
public class RetailBankCustomer
     extends BankCustomer {
  private int fees;
}
```

```
@Entity
public class PrivateBankCustomer
        extends BankCustomer {
  private String eliteOffer;
}
```

Anfragen in JPA: JPQL

Anfragen mit JPQL

- Java Persistence Querying Language
- Anfragesprache in Analogie zu SQL … aber: Operiert auf Entity Modell, nicht DB-Modell!
  - Entities & Fields statt Tables & Columns
  - Wird von JPA Provider in SQL übersetzt
- Beispiel:

```
Query query =
  em.createQuery("select c from BankCustomer c join c.accounts a");
```

Fällt Ihnen an dieser Anfrage etwas auf (im Vgl. mit SQL)?

JPQL: Beispiele

```
select a from BankAccount a
```

Alias

```
select a from BankCustomer c join c.accounts a
```

Join nur via Relations

```
select distinct a.id, a.balance from BankAccount a
```

Projektion (Vector)

```
select a from BankAccount a
where a.balance >= 0 and a.balance <= 1000
```

Filter

```
select a from BankAccount a order by a.balance desc
```

Sortierung

Query Parameters

- Positional Parameters

```
select a from BankAccount a where
a.customer.name like ?1 and a.balance >= ?2
```

- Named Parameters

```
select a from BankAccount a where
a.customer.name like :name and a.balance >= :lower
```

Parameter setzen

- Parameter setzen
  - Position: `query.setParameter(1, "Bob")`
  - Named: `query.setParameter("lower", 100)`
  - Auch Entities sind als Parameter möglich
- Verhinderung der SQL Injection

Dynamic Queries

- Query zur Laufzeit gebaut und geprüft

```
Query query = em.createQuery(
 "SELECT c FROM BankCustomer c WHERE c.name LIKE :customerName"
);
query.setParameter("customerName", name);
query.setMaxResults(1000);
List<BankCustomer> list = query.getResultList();
```

Named Queries

- Statisch bekannt
  - Können von JPA Provider vorgeparst und optimiert werden
  - @NamedQuery muss bei Entity-Klasse annotiert werden

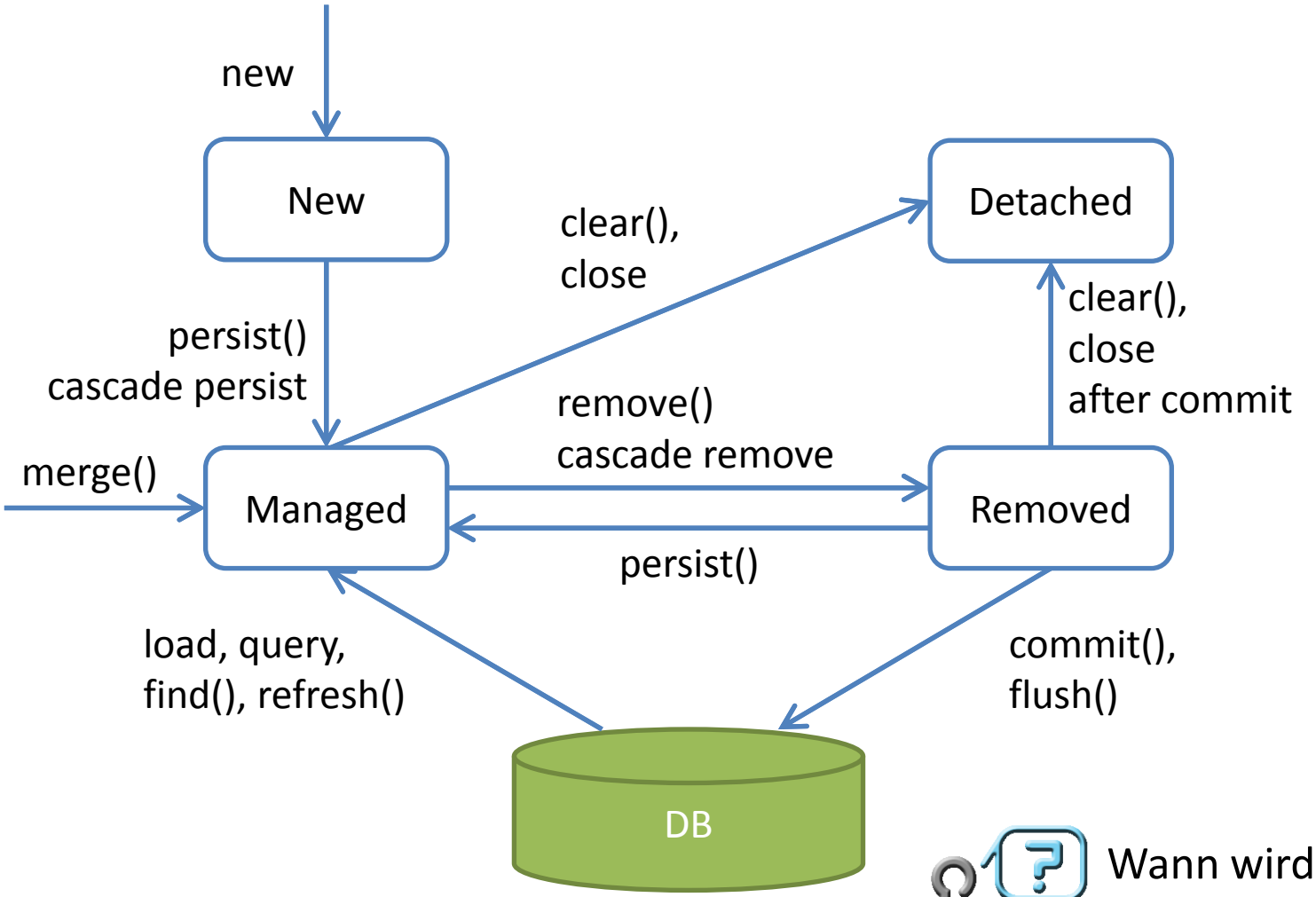
```
@NamedQuery(name = "CustomerSearch", query =
 "SELECT c FROM BankCustomer c WHERE c.name LIKE :customerName")
@Entity
public class BankCustomer { … }

…

Query query = em.createNamedQuery("CustomerSearch");
query.setParameter("customerName", name);
```

Kontrolle über Verhalten und Zustände der Entities

Entity Lebenszyklus



Wann wird eine Entity Garbage?

Refresh und Merge

- Zustand neu von DB laden

```
em.refresh(customer);
```

- Fremde Entity in Persistenzkontext mergen
  - Managed Copy als Rückgabe

```
managedCustomer = em.merge(detachedCustomer);
```

Transitive Persistenz

- Alle von persistentem Objekt erreichbaren Objekte sollen wiederum persistent sein
  - Alle anderen Objekte müssen nicht persistent sein
- Keine automatische transitive Persistenz in JPA
  - Entweder explizit in DB allozieren / deallozieren
    - EntityManager.persist(), EntityManager.remove()
  - Oder implizit via kaskadierte Relations
    - Kaskade-Angabe bei Relation-Annotation

Explizite (De-)Allokation

```
customer.addAccount(newAccount);
em.persist(newAccount);

customer.removeAccount(oldAccount);
em.remove(oldAccount);
```

Erreichbar von persistentem Objekt

Nicht mehr erreichbar von persistenten Objekten

Kaskadierte Persistenz

- CascadeType.Persist
  - Referenzierte Objekte werden automatisch persistent

```
@OneToMany(cascade = CascadeType.PERSIST, …)
private Collection<BankAccount> accounts = new ArrayList<>();
```

```
BankAccount newAccount = new BankAccount()
customer.addAccount(newAccount);
```

newAccount wird implizit persistent

Kaskadiertes Löschen

- CascadeType.Remove
  - Löschen des Holder-Objekts bewirkt Entfernen der referenzierten Objekte

```
@OneToMany(cascade = CascadeType.Remove)
private Collection<BankAccount> accounts = new ArrayList<>();
```

```
em.remove(oldCustomer);
```

oldAccount wird implizit auch gelöscht

Relations mit Abhängigkeiten

- Aggregation


```
@OneToMany(cascade = CascadeType.PERSIST, …)
private Collection<BankAccount> accounts = new ArrayList<>();
```

- Komposition


```
@OneToMany(cascade = { CascadeType.PERSIST, CascadeType.REMOVE }, …)
private Collection<BankAccount> accounts = new ArrayList<>();
```

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Diskussion: JPA und Implementationen

- JPA-Spezifikation:
  - Query-Sprache und Annotationen binden ‚bindet' Domain an Persistenzschicht
  - RDBMS könnten noch mehr abstrahiert sein
  - Bedarf nach besserer Konfiguration, u.a.
    - für DB-spezifische Typen und Funktionalitäten
    - Field Access ok, aber manchmal Property Access
- Implementationen allgemein:
  - +: mehrere Implementationen
  - +/-: Tools erst am Kommen und ihrem Standard voraus
  - -: Noch nicht ausgereift, z.B. benutzerdef. Datentypen

Diskussion: Fazit

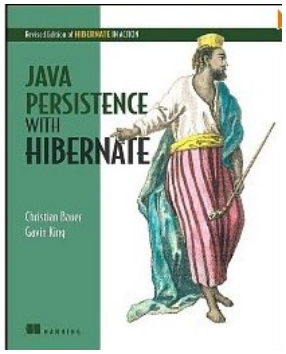
- Erfahrungen mit DataNucleus:
  - JDO bietet bessere OO-Kompatibilität als JPA
  - Aber kein lazy-loading von grossen Recordsets
  - Two-man-show
- Erfahrungen mit Hibernate:
  - konform zu JPA und EJB 3.0, jedoch auch eigene Query-Sprache
  - Ok, falls RDBMS gesetzt und Standard nicht im Vordergrund
  - Verbreitet, aber mehr Dependencies als z.B. EclipseLink
  - Breite Palette von unterstützten DBMS

Lernziele Heute

- ✓ Sie sind in der Lage, den JPA auch für komplexe OR-Mappings einzusetzen.
- ✓ Sie kennen die Funktionsweise der kaskadierten Persistenz und die Verwaltung der Entities in JPA.
- ✓ Sie wissen, wie man Entity-Identitäten und Vererbung in JPA abbildet.
- ✓ Sie haben einen Überblick über JPQL.

Lesestoff

- JPA Tutorial
  - http://docs.oracle.com/javaee/5/tutorial/doc/bnbpz.html
- JPA 2.0 Specification (JSR317)
  - http://jcp.org/aboutJava/communityprocess/final/jsr317
- EclipseLink (JPA Reference Implementation)
  - http://wiki.eclipse.org/EclipseLink
- C. Bauer and G. King. Java Persistence with Hibernate, Manning, 2007
- D. Röder. JPA mit Hibernate, Java Persistence API in der Praxis, Entwickler Press, 2010



Selbststudium

Entity-Identität

- Annotation @Id
  - Für jede Entity eine ID-Property
  - Bildet auf Primary Key in DB ab
- Generierung der Identity
  - Annotation @GeneratedValue bei Id-Property
  - Generiert durch JPA-Instanz beim INSERT neuer Rows in DB
- Verschiedene Strategien für Generierung der ‚id‘
  1. AUTO
  2. IDENTITY
  3. SEQUENCE
  4. TABLE

Generierungstyp Identity

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long accountId;
```

```
CREATE TABLE bankaccount (
  accountid SERIAL NOT NULL,
  CONSTRAINT account_pkey PRIMARY KEY (accountid)
  …
)
```

Generierungstyp Sequence

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
                generator = "BankCustGen")
@SequenceGenerator(name = "BankCustGen",
                   sequenceName = "CustomerIdSeq",
                   allocationSize=1)
private long customerId;
```

Explizites Sequence Objekt in DB

Table BankCustomer

customerId BIGING	name TEXT
1	Bob

```
CREATE SEQUENCE customeridseq;
```

Generierungstyp Table

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE,
      generator ="CustomerGen")
@TableGenerator(name ="CustomerGen", table ="KeyTable",
      pkColumnName ="KeyName",
      valueColumnName ="KeyValue",
      pkColumnValue ="CustomerKey")
private long customerId;
```

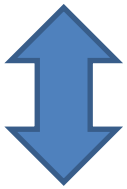


Tabelle zur Key-Verwaltung

KeyTable

KeyName	KeyValue
CustomerKey	100
AccountKey	3214

Letzter Primary Key für BankCustomer

Änderungen an Relations

- Beziehungen lassen sich normal zugreifen
  - Eager vs. Lazy Load Verhalten bei Zugriff
  - Änderungen werden bei Commit in DB gespeichert

```
customer.getAccounts().add(newAccount);

customer.getAccounts().remove(oldAccount);

oldAccount.setCustomer(newCustomer)
```

Inkonsistente Bidirektionalität

- Achtung: Bidirektionale Relationen sind bei Änderungen durch JPA nicht synchronisiert
  - Trotz Angabe von z.B. `@OneToMany(mappedBy="customer")`
  - Nur nach dem Laden konsistent

```
customer.getAccounts().add(newAccount);
if (newAccount.getCustomer() != customer) {
  System.out.println("Bidirectional relation out of sync!");
}
```

Bidirectional Sync (1)

```
public class BankAccount {
  …
  public void setCustomer(BankCustomer newCustomer) {
    BankCustomer oldCustomer = this.customer;
    this.customer = newCustomer;
    if (newCustomer != null && !newCustomer.containsAccount(this)) {
      newCustomer.addAccount(this);
    }
    if (oldCustomer != null && oldCustomer.containsAccount(this)) {
      oldCustomer.removeAccount(this);
    }
  }
}
```

Studieren Sie den Code. Was bezweckt:

- Null Test?
- Contains Test?

Beispiel: Bidirectional Sync (2)

```
public class BankCustomer {
  …
  public void addAccount(BankAccount account) {
    this.accounts.add(account);
    if (account.getCustomer() != this) {
      account.setCustomer(this);
    }
  }
  …
}
```

Verhindere Endlosrekursion

Interne Collection nicht herausgeben

JPA Transaktionen

- Änderungen in expliziten Transaktionen

```
EntityManager em = factory.createEntityManager();
em.getTransaction().begin();
// ...
// do something... make changes
// ...
em.getTransaction().commit(); // or rollback();
```

Transaktionsende

- Nach Commit
  - Änderungen in DB
  - Entity-Zustände nicht von DB refreshed
- Nach Rollback
  - Änderungen nicht in DB
  - Entities sind detached (behalten Zustand)

Transaction Isolation

- JPA basiert auf READ COMMITTED Isolation Level!

Welche Fehlerphänomene können damit auftreten?

- Je nach JPA Provider andere Levels konfigurierbar
  - Beispiel Hibernate (in persistence.xml)
  - Anderer Default DB Isolation Level nützt nicht immer
    - EclipseLink führt Reads nicht über gleiche Connection wie Writes aus

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

JPA Transaktionen: Explizites Locking

- Methode lock() beim EntityManager
  - Soll Dirty Reads und Lost Updates ausschliessen

```
BankAccount account = em.find(…);
em.lock(account, LockModeType.PESSIMISTIC_WRITE);
account.incBalance(100);
```