



Java Persistence API 2.0

Expertenkreis Java, 24.06.2010

Dirk Weil, GEDOPLAN GmbH



Warum JPA?

- ≡ Aufgabenstellung:
 - ≡ Speichern und Laden von Java-Objekten
 - ≡ Mapping OO ↔ RDBMS

- ≡ Lösungsansätze:
 - ≡ JDBC
 - ≡ Entity-EJBs (bis EJB 2.1)
 - ≡ O/R-Mapper

```
public class Country
{
    private String isoCode;
    private String name;

    public Country(String isoCode,
                   String description)
    {
        ...
    }

    public String getIsoCode()
    {
        return this.isoCode;
    }

    public void setIsoCode(String isoCode)
    {
        this.isoCode = isoCode;
    }
}
```



Warum JPA?

- ≡ Konventionelle Lösung
 - ≡ Direktes JDBC

```
Country country = new Country(...);  
...  
Connection connection = JdbcUtil.getConnection();  
  
PreparedStatement statement = connection.prepareStatement(  
    "insert into Country(isoCode,name) values (?,?)");  
statement.setString(1, country.getIsoCode());  
statement.setString(2, country.getName());  
  
statement.executeUpdate();  
  
connection.commit();  
...
```



Warum JPA?

- ≡ JPA-Lösung
 - ≡ Normale Java-Klasse (POJO) mit Annotationen

```
@Entity
public class Country
{
    @Id
    private String isoCode;
    private String name;

    ...
}
```

```
Country country = new Country(...);
...
EntityManager em
    = entityManagerFactory.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

em.persist(country);

tx.commit();
```



Warum JPA?

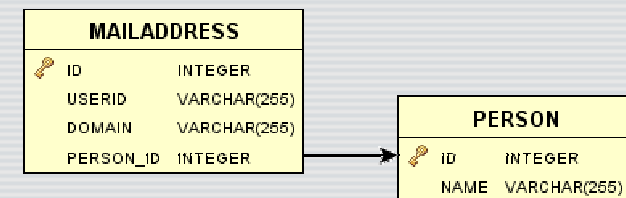
☰ Relationen (1:1, 1:n, n:m)

```
@Entity
public class Person
{
    @Id
    public Integer      id;
    public String      name;

    @OneToMany
    @JoinColumn(name = "PERSON_ID")
    public List<MailAddress> mailAddresses
        = new ArrayList<MailAddress>();
}
```

1 → n

```
@Entity
public class MailAddress
{
    @Id
    public Integer id;
    public String  userId;
    public String  domain;
}
```





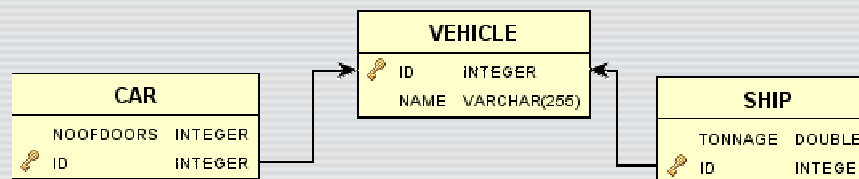
Warum JPA?

≡ Vererbungs- beziehungen

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Vehicle
{
    @Id
    private Integer id;
    private String name;
```

```
@Entity
public class Car extends Vehicle
{
    private int noOfDoors;
```

```
@Entity
public class Ship extends Vehicle
{
    private double tonnage;
```



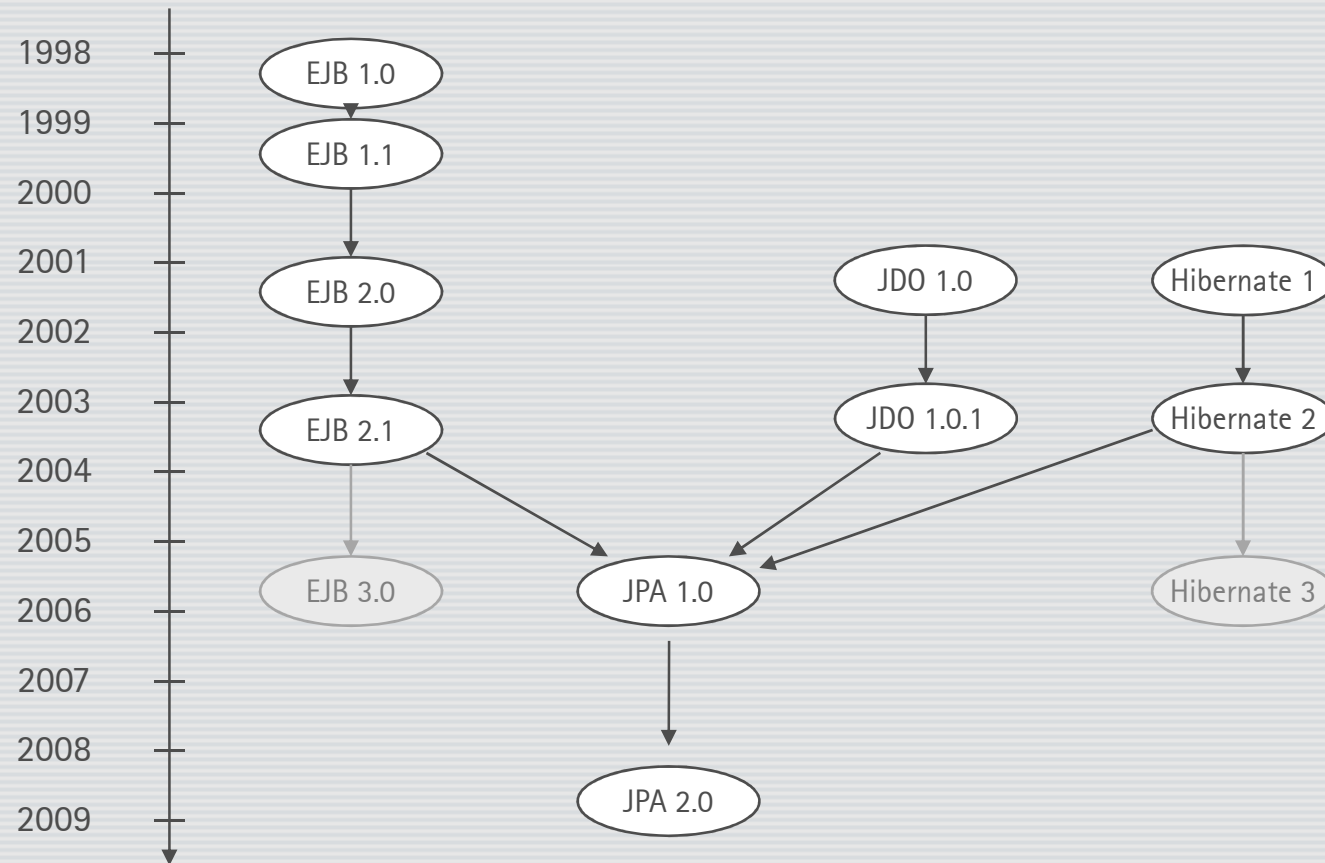


Warum JPA?

- ≡ Anforderungen an O/R-Mapper
 - ✓ Feld-Zuordnung
 - ✓ Erstellen der SQL-Befehle
 - ✓ ID-Generierung
 - ✓ Navigation über Relationen
 - ✓ Abbildung von Vererbung
 - ✓ Verwaltung kompletter Objekt-Graphen
 - ✓ Verbindungsverwaltung
 - ✓ Transaktionsverwaltung
 - ✓ Caching
 - ✓ Schemagenerierung



Entwicklung des Standards 'JPA'





Detached Entities

- ≡ Managed Entities können vom Entity Manager gelöst werden
 - ≡ mittels `clear` oder `detach`
 - ≡ mittels `rollback`
 - ≡ durch Schließen des Entity Managers
 - ≡ durch Serialisierung/Deserialisierung

```
EntityManager em = ...;

String isoCode = "DE";
Country country = em.find(Country.class, isoCode);

em.detach(country);

...
```



Mapping-Annotationen und Defaults

- ≡ Configuration by Exception → Meist gute Defaults vorhanden
- ≡ Dennoch: Werte angeben
 - ≡ Tabellen- und Spaltennamen
 - ≡ Zugriffstyp
 - ≡ ...

```
@Entity
@Table(name="COUNTRY")
@Access(AccessType.FIELD)
public class Country
{
    @Column(name="ISO_CODE")
    @Id
    private String isoCode;
```



Queries

- ≡ Abfragesprache: JPQL
- ≡ Ähnlich SQL, jedoch objektorientiert
- ≡ Java-Interface: `TypedQuery<E>` (bis JPA 1.0 nur `Query`)
- ≡ Ausführung mit `getSingleResult` bzw. `getResultList`

```
TypedQuery<Country> query = em.createQuery(  
    "select c from Country c where c.carCode='D'", Country.class);  
  
Country c = query.getSingleResult();
```

```
TypedQuery<Country> query = em.createQuery(  
    "select c from Country c where c.name like 'D%'", Country.class);  
  
List<Country> l = query.getResultList();
```



Orphan Removal

≡ "Garbage Collection" für abhängige Objekte

```
@Entity
public class Order
{
    @Id
    @GeneratedValue
    private Integer      id;

    @OneToMany(mappedBy = "order", orphanRemoval = true)
    private List<OrderLine> orderLines = new ArrayList<OrderLine>();
}
```

```
@Entity
public class OrderLine
{
    @Id
    @GeneratedValue
    private Integer id;
    private String  name;
    private int     count;

    @ManyToOne
    private Order   order;
}
```



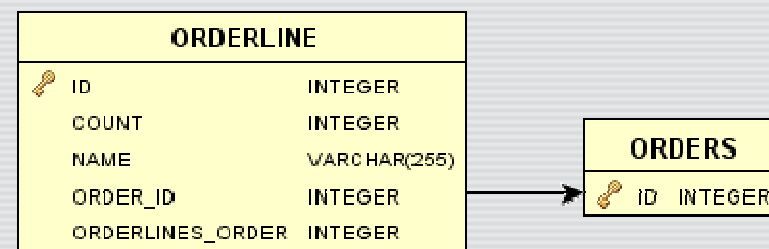
Anordnung von Relationenelementen

- ☰ Kann durch ein Feld der referenzierten Entity erzwungen werden

```
@OneToMany(mappedBy = "order")  
@OrderBy("name")  
private List<OrderLine> orderLines = new ArrayList<OrderLine>();
```

- ☰ Alternative: Persistente Ordnung mittels zusätzlicher Spalte

```
@OneToMany(mappedBy = "order")  
@OrderColumn(name = "ORDERLINES_ORDER")  
private List<OrderLine> orderLines = new ArrayList<OrderLine>();
```





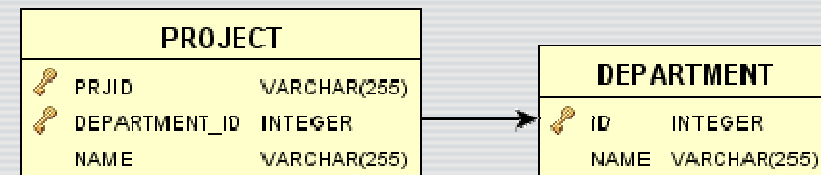
Relationen als Id-Attribute (Derived Identity)

- ≡ Häufig bei zusammengesetzten Schlüsseln
- ≡ Id-Attribut stellt Relation dar

```
@Entity
@IdClass(ProjectId.class)
public class Project
{
    @Id
    @ManyToOne
    private Department department;

    @Id
    private String prjId;
    ...
}
```

```
public class ProjectId
    implements Serializable
{
    public Integer department;
    public String prjId;
    ...
}
```



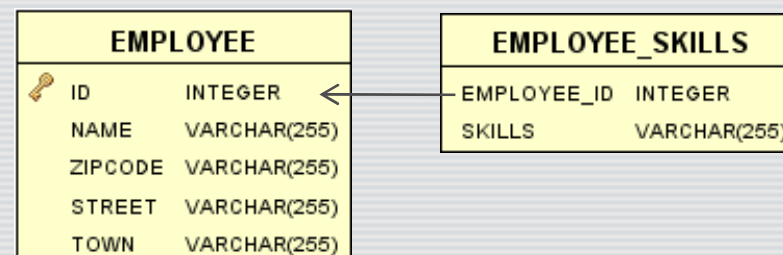


Collections von einfachen Typen oder einbettbaren Objekten

- ≡ Attribute vom Typ `Collection<E>` oder davon abgeleitet
- ≡ Einfache Elementtypen oder Embeddables

```
@Entity
public class Employee
{
    ...
    @ElementCollection(fetch = FetchType.EAGER)
    private List<String> skills = new ArrayList<String>();
    ...
}
```

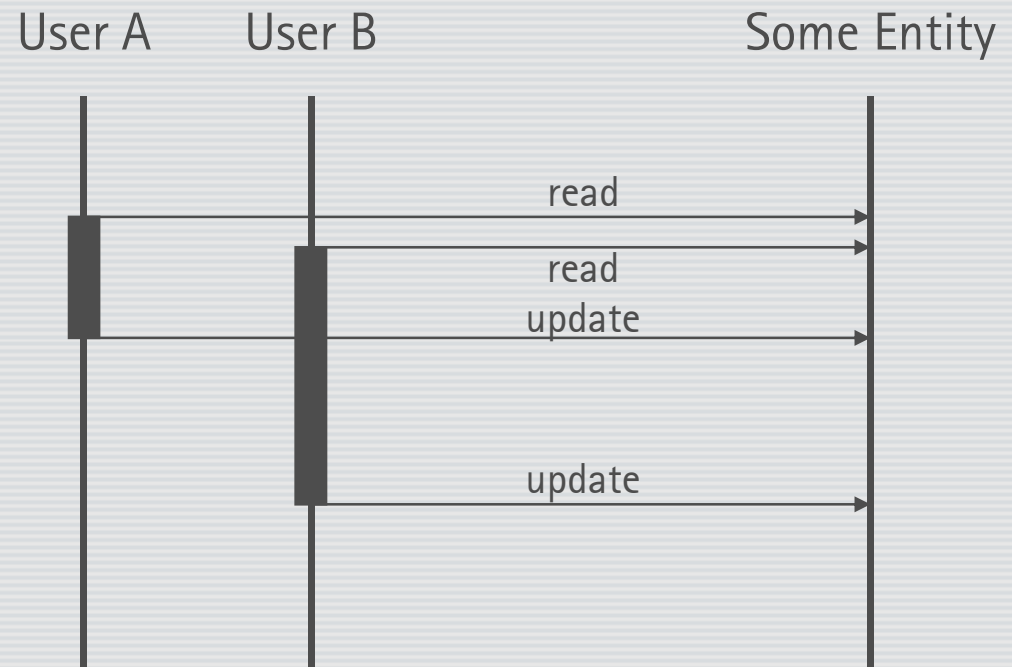
- ≡ Abbildung auf Zusatztabelle





Locking

- ≡ Problemstellung:
Gleiche Daten werden
parallel verändert



- ≡ Abhilfe:
 - ≡ Optimistic Locking (mittels Versionsattribut)
 - ≡ Pessimistic Locking (durch Sperren in der DB)



Locking

- ≡ Pro Objekt anwendbar
 - ≡ beim Lesen
 - ≡ für bereits gelesene Objekte

```
em.lock(someEntity,  
        LockModeType.PESSIMISTIC_WRITE);
```

LockModeType .	Bedeutung
NONE	Keine Sperren nutzen
OPTIMISTIC	Optimistic Locking benutzen
OPTIMISTIC_FORCE_INCREMENT	dito, aber mit Erhöhung des Versionsattributs
PESSIMISTIC_READ	Shared Lock benutzen
PESSIMISTIC_WRITE	Exclusive Lock benutzen
PESSIMISTIC_FORCE_INCREMENT	dito, aber mit Erhöhung des Versionsattributs



Criteria Queries

≡ Problem: Keine Korrektheitskontrolle von JPQL zur Compilezeit, z.B.

≡ falsche Schlüsselwörter

```
select c fron Cocktail c
```

≡ unvollständige Statements

```
select c from Cocktail
```

≡ falsche Attributnamen

≡ Typkonflikte

```
select c from Cocktail c where c.nam=:name
```

≡ Criteria Query API

≡ objektorientiert

≡ stellt Vollständigkeit sicher

≡ typsicher



Criteria Queries

```
// "select c from Cocktail c where c.name=:name"

CriteriaBuilder builder = em.getCriteriaBuilder();

// Criteria Query für Ergebnistyp erzeugen
CriteriaQuery<Cocktail> cQuery = builder.createQuery(Cocktail.class);

// Projektionsvariablen erzeugen (FROM-Klausel)
Root<Cocktail> c = cQuery.from(Cocktail.class);

// Selektion angeben (SELECT-Klausel)
cQuery.select(c);

// Bedingung erstellen und der Query hinzufügen
Predicate hatNamen = builder.equal(c.get("name"), name);
cQuery.where(hatNamen);

// Query ausführen
TypedQuery<Cocktail> q = em.createQuery(cQuery);
List<Cocktail> found = q.getResultList();
```



Statisches JPA-Metamodell

- Metamodell-Klasse $E_$ zu jeder persistenten Klasse E

```
@Entity
public class Cocktail
{
    @Id
    @GeneratedValue
    private Integer    id;

    private String    name;

    @ManyToMany
    private Set<Zutat> zutaten = new HashSet<Zutat>();
}
```

```
@StaticMetamodel(Cocktail.class)
public abstract class Cocktail_
{
    public static volatile SingularAttribute<Cocktail, Integer> id;
    public static volatile SingularAttribute<Cocktail, String> name;
    public static volatile SetAttribute<Cocktail, Zutat> zutaten;
}
```



Criteria Queries / Statisches JPA-Metamodell

```
// "select c from Cocktail c where c.name=:name"

CriteriaBuilder builder = em.getCriteriaBuilder();

// Criteria Query für Ergebnistyp erzeugen
CriteriaQuery<Cocktail> cQuery = builder.createQuery(Cocktail.class);

// Projektionsvariablen erzeugen (FROM-Klausel)
Root<Cocktail> c = cQuery.from(Cocktail.class);

// Selektion angeben (SELECT-Klausel)
cQuery.select(c);

// Bedingung erstellen und der Query hinzufügen
Predicate hatNamen = builder.equal(c.get(Cocktail_.name), name);
cQuery.where(hatNamen);

// Query ausführen
TypedQuery<Cocktail> q = em.createQuery(cQuery);
List<Cocktail> found = q.getResultList();
```



Weitere Neuerungen in JPA 2.0

- ≡ Zusätzliche Id-Typen
- ≡ Explizite Access Types
- ≡ Verschachtelte Embeddables
- ≡ Embeddables mit Relationen
- ≡ Unidirektionale 1:n-Relationen
- ≡ Erweiterung von JPQL
- ≡ Ermittlung des Load State
- ≡ Caching
- ≡ ...



Mehr ...

- ≡ im Java-Magazin 4.2010
 - ≡ auch auf www.gedoplan.de → Veröffentlichungen

- ≡ in IPS-Seminaren

- ≡ oder: Fragen Sie!

Java EE 6 ausgepackt Enterprise

Java Persistence API 2.0

Die Java Enterprise Edition steht seit dem Dezember 2009 in der lang erwarteten Version 6 zur Verfügung. In den letzten 3 1/2 Jahren hat es Weiterentwicklungen in nahezu allen Bereichen gegeben. Nachdem wir uns in [1] die aktuellsten Möglichkeiten im Bereich der Enterprise JavaBeans angeschaut haben, wenden wir uns einem zentralen Punkt nahezu aller Anwendungen zu, nämlich der Objektpersistenz.

von Dirk Weil

Für die persistente Ablage von Geschäftsobjekten hat es in der Historie der Enterprise Edition mehrere Ansätze gegeben. Der zunächst eingeschlagene Weg über Entity Beans - Enterprise JavaBeans mit einem Mapping zur Datenbank in ihrem Deployment Descriptor - erwies sich aufgrund der damaligen Komplexität von EJBs als zu aufwändig und stark. Zudem musste man erkennen, dass das EJB-Komponentenkonzept für die Speicherung von Daten überflüssig und eher hinderlich ist.

Der zweite Versuch - Java Data Objects - hat einen Wandel von schwerwichtigen Komponenten zu einfachen Java-Klassen (POJO) ein. Trotzdem hat es JDO nicht richtig bis in die Herzen der Entwickler geschafft, vielleicht wegen der damals noch etwas unheimlichen Bytecode Enhancements, mit dem die Klassen mit den notwendigen Persistenzaspekten instrumentiert wurden. Insofern war es nachvollziehbar, dass die strategische Ausrichtung der Enterprise Edition zwar durch JDO beeinflusst wurde, darüber hinaus aber insbesondere

Frameworks wie dem äußerst populären Hibernate zur Entwicklung eines neuen Standards genutzt wurden: Java Persistence API. Die aktuelle Enterprise Edition umfasst nun JPA in der neuen Version 2.0. Damit sind viele Neuerungen und Ergänzungen verbunden, die wir im Folgenden darstellen wollen.

■ **Artikelserie: Java EE 6 ausgepackt**
Teil 1: EJB 3.1
Teil 2: JPA 2.0

www.java-magazin.de javamagazin 4/2010 | 69