

Java Persistence Query Language (JPQL)

Einleitung

Java Persistence Query Language (JPQL)

- Lehnt sich an die Structured Query Language (**SQL**) an.
- Agiert aber anders als SQL **auf der Objektseite** und nicht auf der Datenbankseite.
- Möglich sind z.B.
 - Herausfiltern von Datensätzen mit Where
 - Joins
 - Aggregationsfunktionen
 - Sortieren
 - Bulk-Updates und Deletes

Wie lassen sich JPQL-Queries absetzen?

Der Entity-Manager hält folgende Methoden bereit:

- **createQuery()**
 - Diese haben wir schon benutzt, um Personen und Sprachen aus der Datenbank zu beziehen.
- **typisierte createQuery()**
 - Erweiterung um Typsicherheit zu gewährleisten
- **createNamedQuery()**
 - Die Query selbst wird aus einer Annotation entnommen.
- **createNativeQuery()**
 - Nutzung von SQL zur direkten Datenbankabfrage

Nicht-typisierte createQuery()-Abfragen

- Haben wir schon benutzt:

```
@SuppressWarnings("unchecked")
```

```
public Collection<Person> findAll() {
```

```
    Query query = em.createQuery(  
        "SELECT p FROM Person p");
```

```
    Collection<Person> collection;
```

```
    collection = (Collection<Person>)  
        query.getResultList();
```

```
    return collection;
```

```
}
```

Unser Problem: Wir wollen
Typisierung!

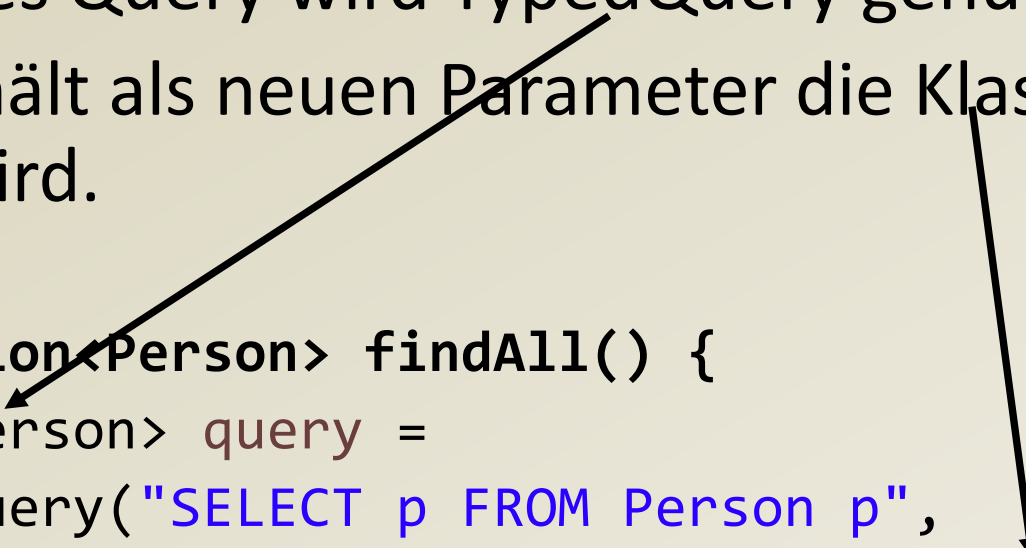
Query wird an
createQuery()
übergeben.

Query wird
ausgeführt.

Typisierte createQuery()-Abfragen

- Statt des Interfaces Query wird TypedQuery genutzt
- createQuery() erhält als neuen Parameter die Klasse, die zurückgegeben wird.

```
public Collection<Person> findAll() {  
    TypedQuery<Person> query =  
        em.createQuery("SELECT p FROM Person p",  
                        Person.class);  
  
    Collection<Person> collection;  
    collection = query.getResultList();  
    return collection;  
}
```



Benannte Abfragen mit createNamedQuery()

- Der JPQL-Code verstreut sich häufig über die gesamte Klasse.
- Deshalb kann der JPQL-Code in den Kopf der Entity-Klasse per Annotation ausgelagert werden:

```
@NamedQuery(name="dername", query="SELECT p FROM Person p")
```
- Wenn es mehrere Named Queries gibt:

```
@NamedQueries({  
    @NamedQuery(name="dername", query="..."),  
    @NamedQuery(name="zweitername", query="...")  
})
```
- Innerhalb einer Methode, auch in einer anderen Klasse, kann er wie folgt abgefragt werden:

```
Query query = em.createNamedQuery("dername");  
  
TypedQuery<Person> query =  
    em.createNamedQuery("dername", Person.class);
```

Native SQL-Abfragen

- Sollte nur dann genutzt werden, wenn JPQL-Anfragen nicht mehr ausreichen.
- Es können auch komplizierte SQL-Anfragen an die Datenbank übermittelt werden.
- Zwei Arten:
 - Query `query` = `em.createNativeQuery("SQL statement");`
Die Ergebnisse werden hier als eine Collection an Objekt-Arrays zurückgegeben.
 - Query `query` = `em.createNativeQuery("SQL statement", Person.class);`
 - Es wird in die Klasse versucht zu mappen.
 - Die Felder müssen allerdings in dieser Klasse verfügbar sein.
 - Falls nicht, muss ein Mapping genutzt werden.

@SqlResultSetMapping

```
@SqlResultSetMapping(name="querybeispiel",
    entities={
        @EntityResult(entityClass=Person.class, fields={
            @FieldResult(name="id", column="id_in_result"),
            @FieldResult(name="name", column="name_in_result")
        })),
    columns={
        @ColumnResult(name="sum")}
    )
```


@NamedNativeQuery

- Für native Abfragen existiert als Gegenstück zu `@NamedQuery` die `@NamedNativeQuery`
- Diese Annotation ist identisch zu der `@NamedQuery` aufgebaut.
- Es wird dieselbe Methode bei der Abfrage verwendet
`Query query = em.createNamedQuery("dername");`
- Zusätzlich kann auch die Klasse angegeben werden, die zurückgegeben wird:
`TypedQuery<Person> query = em.createNamedQuery("dername", Person.class);`

Entgegennahme der Resultate

- **getResultList()**
 - Rückgabe der kompletten Liste
- Paginierung möglich
 - **setMaxResults(100)**: Maximal zunächst 100 Zeilen lesen.
 - **setFirstResult(100)**: Beim nächsten Mal kann ab Eintrag 101 weitergearbeitet werden.
(setFirstResult() beginnt von 0 an zu zählen)
- **getSingleResult()**
 - Wenn nur ein Ergebnis erwartet wird.
 - Wenn mehrere Ergebnisse zurückgegeben werden, wird eine **NonUniqueResultException** zurückgegeben.
 - Wenn kein Ergebnis zurückgegeben wird, erhalten wir eine **NoResultException**.

Parameter

- Innerhalb der Statements können mit einem Doppelpunkt Parameter angegeben werden:

```
@NamedQuery(name="dername", query="select p from  
Person p where gehalt > :mingehalt")
```

- Diese werden befüllt mit

```
query.setParameter("mingehalt", 1000);  
query.setParameter(1, 1000); (Wichtig ab Position 1,  
                             nicht ab 0)
```

- Es können auch temporale Werte übergeben werden.
 - Hier kann mit einem dritten Parameter entschieden werden, ob TemporalType.**DATE**, TemporalType.**TIME** oder TemporalType.**TIMESTAMP** genommen werden soll.
- Nutzen mit Fluent-Interface (sprechende Schnittstelle) möglich.

Java Persistence Query Language (JPQL)

Die Sprache selbst

Select ... From ...

Die normalen **Select ... From ... Where**-Klauseln haben wir schon kennengelernt:

```
select p from Person p where p.vorname = :vn
```

Zu sehen ist, dass anders als bei SQL-Statements ...

- ... das ***** hier durch die **p**-Variable für die **Person** ersetzt ist,
- ... die **p**-Variable im **from-Statement** an die **Person** gebunden wird und
- ... im **where-Statement** verwendet werden kann.

Implizite Joins

- Joins können **implizit** durchgeführt werden, indem die JPA angewiesen wird über Referenzen in Objekten zu springen:

```
select p.vorname, p.nachnamen,  
p.adresse.plzOrt from Person p
```

- Erzeugt bei der Abfrage mit `getResultList()` eine Liste an Objekt-Arrays, welche jeweils **vorname**, **nachnamen** und **plzOrt** beinhalten.

Explizite Joins

- Überall dort, wo implizite Joins nicht durchgeführt werden können, können explizite Joins stehen:

```
select p, adr from Person p join p.adresse adr
```

- Erzeugt bei der Abfrage mit `getResultList()` eine Liste an Objekt-Arrays, welche jeweils **das Personobjekt** und **das Adressobjekt** beinhalten.
- Statt eines einfachen **Inner Join** kann natürlich auch ein **Left Join** verwendet werden.

Fetch-Joins

- Wir hatten gelernt, dass wir bei Beziehungen entweder `fetch=FetchType.LAZY` oder `fetch=FetchType.EAGER` angeben können.
- Das LAZY-Verhalten, welches das Default Verhalten für 1:N und N:M-Beziehungen ist, ist entweder mit `fetch=FetchType.EAGER` oder mit **fetch-Joins** überschreibbar:

```
select p from Person p join fetch p.emailaddresses
```

- Es werden die Emailadressen mit einem SQL-Statement mit geladen.

Lösen der N+1-Problematik mit dem Fetch-Join

- Das Umsetzen von LAZY auf EAGER mit Hilfe der Annotation kann genutzt werden, wenn die Liste **direkt** an dem zu ladenden Objekt hängt.
- Jedoch können an der Liste **weitere Listen** hängen.
 - Es existieren also beispielsweise zwei hintereinander geschaltete 1:N-Beziehungen
- Hier können nur die JPQL-**fetch-Joins** Abhilfe schaffen, sonst müssen **1 + n Abfragen** an die Datenbank geschickt werden.
- Vorsicht: Joins können sehr große Datenmengen erzeugen.

Bulk Update und Bulk Delete

- Für ein Löschen oder Update müssten die Objekte einzeln aus der Datenbank geholt werden und dann verändert werden.
- Mittels SQL lassen sich viele Daten auf der Datenbank gleichzeitig aktualisieren oder löschen.
- In JPQL ist eine ähnliche Funktionalität eingebaut: **Bulk Update** und **Bulk Deletes**:

```
Query query = em.createQuery  
    ("update Person p set p.nachnamen = 'Meier' where  
     p.nachnamen = 'Mustermann'");  
int resultat = query.executeUpdate();
```

```
Query query = em.createQuery  
    ("delete from Person p where p.vorname =  
     'Max'");  
int resultat = query.executeUpdate();
```