

# ***Objektorientierte Datenbanken***

Vorlesung 5  
Sebastian Iwanowski  
FH Wedel

**JDO: 3. Teil: JDOQL (1. Teil)**

# Datenbankanfragemöglichkeiten ohne Anfragesprache

- a) Extraktion eines einzelnen Objects (spezifiziert durch Name / ID)
- b) Extraktion des Extents einer Klasse

**Ungeeignet für das Problem:**

**Finde Objekte mit bestimmten Eigenschaften !**

- a) Hoher Programmieraufwand
- b) Hoher Hauptspeicherbedarf

# Ziele einer Objektanfragesprache

Beispiele: OQL, **JDOQL**, HQL, VQL

**1) gezielte Extraktion von Objekten mit bestimmten Eigenschaften**

mehr Programmierkomfort, mehr Effizienz

**2) SQL-ähnliche Abfragemöglichkeiten ohne detaillierte Java-Kenntnisse**

politischer und sozialer Grund

mehr Abfragekomfort, mehr Effizienz

*Für JDOQL in Version 1.0.1 galt nur das 1. Ziel*

*In Version 2.0 wurde auch das 2. Ziel realisiert*

# Wesentliche Merkmale von JDOQL-Queries

- Jede Query bezieht sich meistens auf genau eine Klasse
- Jede Query sucht aus einer Menge von Elementen dieser Klasse diejenigen Elemente, die eine bestimmte Eigenschaft erfüllen

## 2 Anfragetypen:

### 1) Filter-Queries (aus Version 1.0.1, auch in 2.0 verfügbar)

- Die gesuchte Eigenschaft der Objekte wird durch einen Booleschen Ausdruck bestimmt (*Filter* genannt)

### 2) Select-Queries (nur in 2.0 verfügbar)

- Die gesuchte Eigenschaft der Objekte wird durch eine SQL-ähnliche Frage bestimmt (eingeleitet durch SELECT)

# Einstiegsbeispiel für JDOQL, Select-Query

```
Properties pmfProps = new java.util.Properties();
pmfProps.put ("javax.jdo.PersistenceManagerFactoryClass",
             "com.versant.core.jdo.BootstrapPMF" );
pmfProps.put ("javax.jdo.option.ConnectionURL", "versant:myDatabase@host" );
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory( pmfProps );
PersistenceManager pm = pmf.getPersistenceManager();
Transaction txn = pm.currentTransaction();

txn.begin();
```

**// Frage formulieren:**

```
String queryString = " SELECT FROM Professoren WHERE Raum(==) `Ü11` ";
```

**// Frageobjekt erzeugen:**

```
Query query = pm.newQuery (queryString );
```

**// Frage stellen:**

```
Collection result = (Collection) query.execute();
```

**// Ergebnis auswerten:**

```
Iteration iter = result.iterator();
```

```
Professoren professor;
```

```
while (iter.hasNext()) {
```

```
    professor = (Professoren) iter.next();
```

```
    print (professor.name);}

}
```

```
txn.commit();
```

# Einstiegsbeispiel für JDOQL, Filter-Query

```
Properties pmfProps = new java.util.Properties();
pmfProps.put ("javax.jdo.PersistenceManagerFactoryClass",
             "com.versant.core.jdo.BootstrapPMF" );
pmfProps.put ("javax.jdo.option.ConnectionURL", "versant:myDatabase@host" );
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory( pmfProps );
PersistenceManager pm = pmf.getPersistenceManager();
Transaction txn = pm.currentTransaction();

txn.begin();
Extent AlleProfessoren = pm.getExtent (Professoren.class, true);
// Frage formulieren:
String queryString = " Raum == `Ü11` ";
// Frageobjekt erzeugen:
Query query = pm.newQuery (AlleProfessoren, queryString );
// Frage stellen:
Collection result = (Collection) query.execute();
// Ergebnis auswerten:
Iteration iter = result.iterator();
Professoren professor;
while (iter.hasNext()) {
    professor = (Professoren) iter.next();
    print (professor.name);}

txn.commit();
```

# Funktionalität von JDOQL-Queries

## Grundfunktionalität:

### Interface PersistenceManager

*public Query newQuery ();*

*public Query newQuery (Class class, Collection elements, String filter);*

*public Query newQuery (Extent elements, String filter);*

*public Query newQuery (String select-query);*

*public Query newQuery (String language, Object language-specific-query);*

### Interface Query

*public void setClass (Class class);*

*public void setCandidates (Collection elements);*

*public void setCandidates (Extent elements);*

*public void setFilter (String booleanExpression);*

*public Object execute ();*      **gibt per default eine Collection zurück**

# Funktionalität von JDOQL-Queries

## Syntax von Select-Queries:

```
select [unique] [ <result> ] [into <result-class-name>]  
[from <candidate-class-name> [exclude subclasses] ]  
[where <filter>]  
[variables <variable -list> ]  
[parameters <parameter-list>]  
[imports <import-list>]  
[group by <grouping-clause> ]  
[order by <ordering-clause>]  
[range <from-range> ,<to-range>]
```

aus: <http://www.solarmetric.com/resources/jdoql-quickref.pdf>



# Funktionalität von JDOQL-Queries

## Merkmale von Filter-Queries:

### Filterdefinition:

- Der Filter entspricht einer Java-Methode, die `boolean` zurückgibt.
- Die Syntax des Filters ist gleich normaler Java-Syntax.
- Es sind fast alle Java-Operationen zulässig (Ausnahmen später).

### Filterauswertung:

- Der Filter wird auf jedes Element der Kandidatenmenge angewandt.
- Die Attribute der Kandidatenelemente können im Filtercode direkt angesprochen werden. Das angesprochene Attribut bezieht sich auf das Element, das gerade ausgewertet wird. Alle Elemente, für die der Filter `true` ergibt, werden im `execute` zurückgegeben.
- Wenn kein Filter explizit definiert wurde, wird die gesamte Kandidatenmenge im `execute` zurückgegeben.

# JDOQL-Queries: Einfaches Beispiel

## Select-Query:

- Finde alle Studenten, die von Iwanowski geprüft wurden:

**select** s **from** Studenten

**where** s.wurdeGeprüft.Prüfer.Name == „Iwanowski“

## Filter-Query ?

Achtung:

Diese Lösung setzt voraus,  
dass wurdeGeprüft ein einwertiges Attribut ist,  
d.h. jeder Student nur eine Prüfung macht.

Für die Abfrage nach mehrwertigen Attributen  
siehe Folie OODB 5-14  
bzw. weitere Beispiele in OODB 6

# Wie kann man erreichen, dass dieselbe Frage für beliebige Professoren gestellt werden kann ?

## Fiktive Select-Query:

- Finde alle Studenten, die von einem bestimmten Professor geprüft wurden:

**select** s **from** Studenten

**where** s.wurdeGeprüft.Prüfer.Name == \$beliebigerProfessorenname\$

## Filter-Query ?

# Funktionalität von JDOQL-Queries

## Filter-Query mit Parametern:

### Interface Query

```
public void declareParameters (String declarationOfParams);
```

```
public Object execute (Object param1);
```

```
public Object execute (Object param1, Object param2);
```

```
public Object execute (Object param1, Object param, Object param3);
```

*Deklaration der formalen  
Parameter für diese Query*

*Ausführung der  
Query mit aktuellen  
Parametern*

- **declarationOfParams: normale Java-Syntax**
- **Typen und Anzahl der Parameter müssen zueinander passen**  
Ausnahme: primitive Datentypen als formale Parameter werden mit zugehörigen Objektdatentypen als aktuelle Parameter aufgerufen, Beispiel: `int` mit `Integer`
- **im Filter dürfen die Parameter wie lokale Variablen benutzt werden**  
(mit den Namen, die in `declarationOfParams` deklariert wurden)

# Funktionalität von JDOQL-Queries

## Filter-Query mit mehr als 3 Parametern:

### Interface Query

```
public void declareParameters (String declarationOfParams);
```

```
public Object executeWithMap (Map mapOfParams);
```

```
public Object executeWithArray (Object[] arrayOfParams);
```

- **Deklaration der formalen Parameter und Verwendung im Filter wie zuvor**
- **Ausführung mit aktuellen Parametern:**

**Vor dem Aufruf mit `execute` muss die aktuelle Parameterliste als `Map` oder `Array` erzeugt werden:**

- **Maps**      **key: Parametername (als `String`)**  
                 **value: eingesetzter Wert vom Typ wie in `declarationOfParams` angegeben**
- **Arrays**     **Elemente: eingesetzte Werte in derselben Reihenfolge wie in**  
                 **`declarationOfParams`**

**Typen und Anzahl der Parameter müssen zueinander passen  
(mit derselben Ausnahme für einfache Datentypen wie zuvor)**

# JDOQL: Bsp. für Attribut, das selbst Collection ist

## SQL-ähnliche Frage:

- Finde alle Studenten, die bei Iwanowski Vorlesung haben:

**select** s **from** Studenten, v **in** s.hört

**where** v.gelesenVon.Name = „Iwanowski“

## Wie geht das mit JDOQL-Select-Queries ?

## Filter-Query ?

# Funktionalität von JDOQL-Queries

## Zugriff auf Attribute, die selbst Collections sind:

Zugriff auf Collection-wertige Attribute im Filtercode mit den Methoden:

- `Collection.isEmpty ()`
  - `Collection.contains (varname)`  
varname muss als Variable deklariert werden.  
varname bindet ein Element der `Collection`  
und sollte in einem zweiten Booleschen Ausdruck weiter eingeschränkt werden
- zur Abfrage der Existenz von Elementen mit bestimmten Eigenschaften*

## Interface Query

*public void declareVariables (String declarationOfVariables);*

- **declarationOfVariables: normale Java-Syntax**
- **im Filter dürfen die Variablen als lokale Variablen benutzt werden (mit den Namen, die in declarationOfVariables deklariert wurden)**