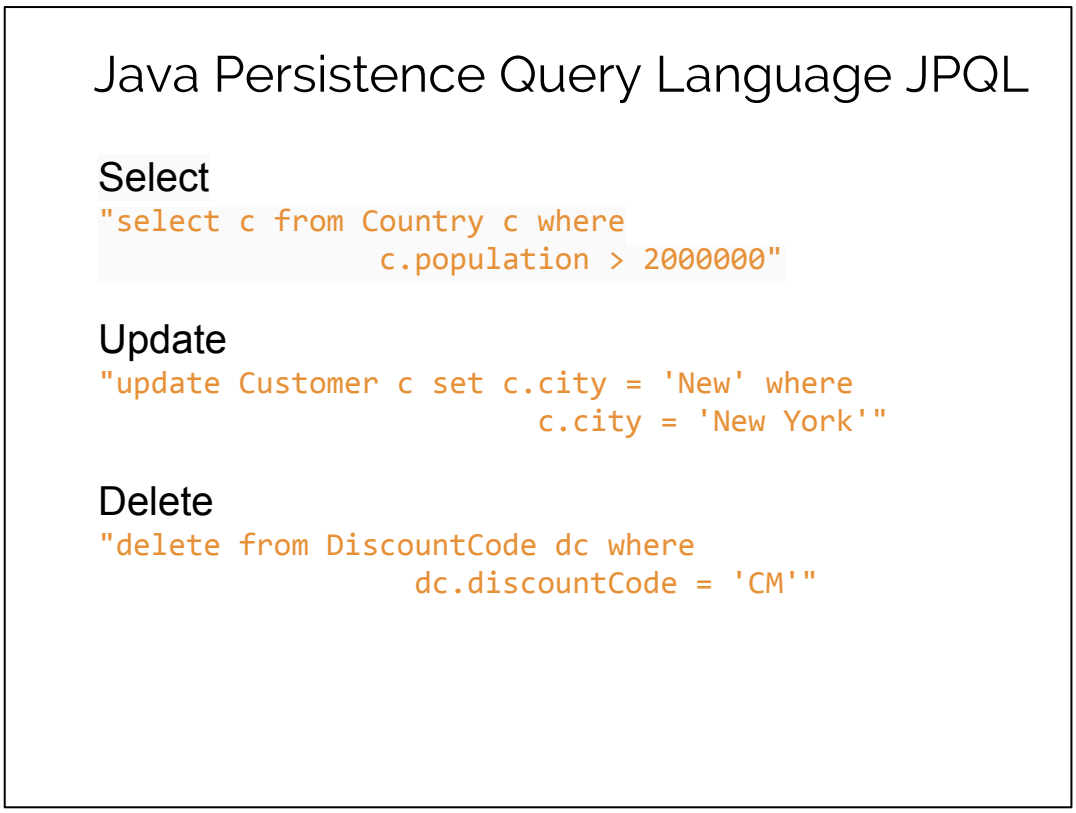
JPA Query API

JPA Slides #4

Content

- Java Persistence Query Language, JPQL
- Persistence Layer Interface



Java Persistence Query Language (JPQL)

- Queries as strings
- Objectified version of SQL, you should feel comfortable …
- Working with (collections of) objects (and types)
- Statements
  - select-statement
  - update-statement
  - delete-statement
- Case insensitive
- Database agnostic (portable)

In the query language, an expression can traverse, or navigate, to related entities. These expressions are the primary difference between the Java Persistence query language and SQL.

- JPQL Queries navigates to related entities, whereas SQL joins tables
- Types and attributes must come from Java classes
  - Not table/column names

JPQL string problematic from design view

- Not type safe (spelling)
  - There is an alternative typesafe query language, the Criteria API, not covered in course
  - NetBeans will generate metadata for Criteria API (Generated Sources). Not used!
- Complexity, looooong strings

- Cut and paste in strings if query very dynamic (don't know until runtime what query we need)
    - Criteria API probably better
- Where to put the JPQL-strings?
    - Where some JPQL aficionado can find and optimize.
        - Java programmer normally not database experts
    - Solutions?
        - No common agreed upon
        - Put named queries on entity classes? More later …
        - Queries in XML, database, …?

Possible to run JPQL from inside NetBeans!

- See code sample jpa_query

JPQL Identification Variables

```
// Assume classes Magazin, Article and Author

// Find all magazines that have articles
// authored by someone with the first name
// "John".
select distinct mag from Magazine mag
   join mag.articles art join art.author auth
   where auth.firstName = 'John'
```

Identification variable(s)

- An identification variable is a valid identifier declared in the from clause of a query
- An identification variable evaluates to a value of the type of the expression used in declaring the variable
  - mag evaluates to type Magazine
  - art evaluates to type Article reachable from Magazin
  - auth evaluates to type Author reachable from Article
  - JQPL is polymorphic: from clause of a query includes not only instances of the specific entity class to which it refers, but all subclasses of that class as well

Possible type errors

// Type error: customerId an object (a customer) customer.id is a Long
"select.... where order.customerId = customer.id"
// Correct, both objects
"select.... where order.customerId = customer"

JPQL Path Expressions

BAD

```
// Assume mag has type Magazin
// Can't use . on collection (i.e. articles)
" ... mag.articles.author ... "
```

SOLUTION

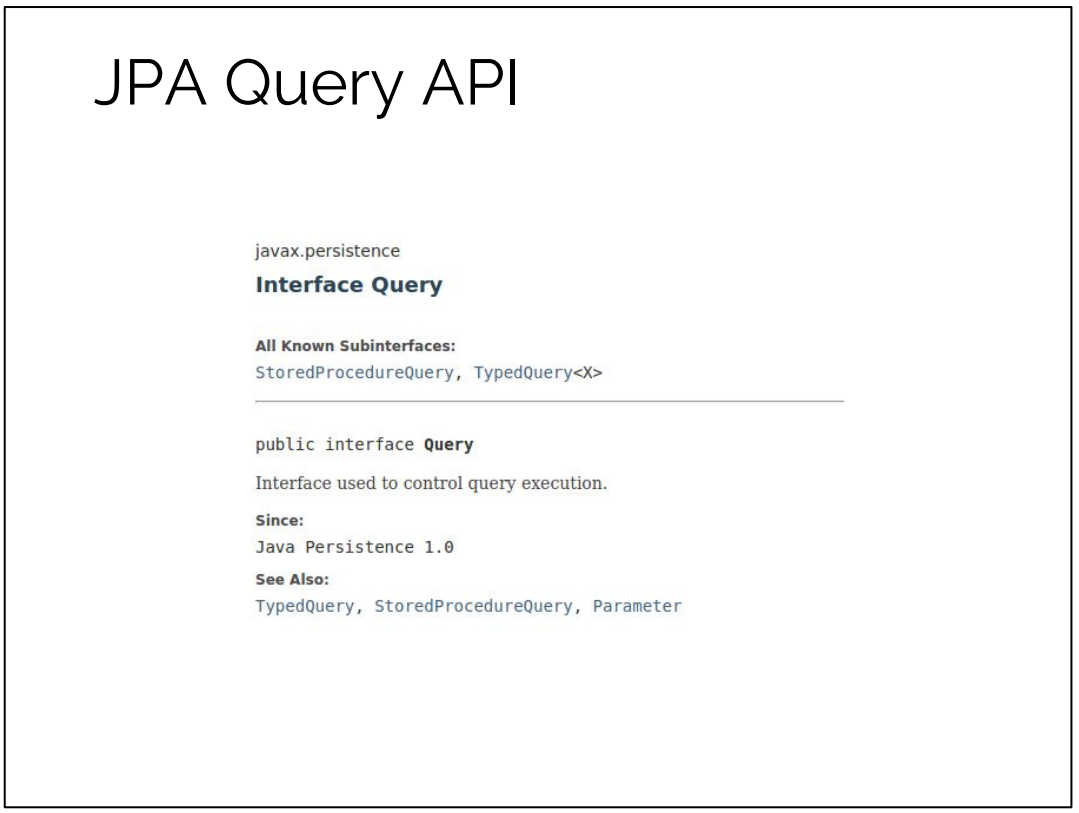
```
// All authors that have any articles
// in any magazines (NOTE ',' before in)
"select distinct art.author
   from Magazin as mag,
   in (mag.articles) art"
```

Extra identification variable

[JPQL Path Expression]

- An identification variable followed by the navigation operator (the dot) and a state-field or association-field is a path expression.
- The type of the path expression is the type computed as the result of navigation
- Having a collection in middle in forbidden
  - Possible with extra identification variable ...
  - ... or join
- Anything in path evaluated to null will be skipped

JPA Query API



javax.persistence

Interface Query

All Known Subinterfaces:
`StoredProcedureQuery, TypedQuery<X>`

`public interface Query`

Interface used to control query execution.

Since:
`Java Persistence 1.0`

See Also:
`TypedQuery, StoredProcedureQuery, Parameter`

Database queries handled by the JPA Query API
- Interface Query, no type in result avoid
- Interface TypedQuery, type in result, prefer!
- Interface StoredProcedureQuery, we don't
- All obtained from EntityManager
- … and some more interfaces

Query API uses embedded query languages
- Native SQL as strings, avoid, not portable
- Criteria API (not used)
- JPQL, we use!

JPA Query API Selections

Single result typed query

```
// Result object managed if inside transaction)
Customer customer = em
    .createQuery("select c from Customer c where
        c.name = 'Olle'",  Customer.class)
    .getSingleResult();
```

Collection result typed query

```
List<Customer> customers = em
    .createQuery("select c from Customer c", Customer.class)
    .setMaxResults(20)
    .getResultList();
```

Will get typed query if supplying a class-parameter

- Always use!
- em in slide is entitymanager instance

Query Parameters

Named Parameters

```
String name = ...;
List<Customer> cs = em.createQuery(
    "select c from Customer c where c.name = :name", Customer.class)
    .setParameter("name", name)
    .getResultList();
```

Positional Parameters

```
String name = ...;
List<Customer> cs = em.createQuery(
    "select c from Customer c where c.name = ?1", Customer.class)
    .setParameter(1, name)
    .setMaxResults(20)
    .getResultList();
```

Named Queries

```
// Queries generated by NetBeans
@Entity
@NamedQueries({
@NamedQuery(name = "Customer.findAll",
    query = "SELECT c FROM Customer c"),
@NamedQuery(name = "Customer.findByName",
    query = "SELECT c FROM Customer c WHERE c.name = :name")})
public class Customer implements Serializable {

}


List<Customer> results = em
    .createNamedQuery("Customer.findByName", Country.class);
    .getResultList();
```

Only on entity classes

- NetBeans can generate

Bulk Operations

Update

```
int nAffectedRows = em
    .createQuery("update Customer c set c.city = :newName
                    where c.city = :oldName")
    .setParameter("newName", "New")
    .setParameter("oldName", "New York")
    .executeUpdate();
```

Delete

```
int nAffectedRows = em
    .createQuery("delete from DiscountCode dc where
            dc.discountCode = :dc")
    .setParameter("dc", 'X')
    .executeUpdate();
```

Applies to entities of a single entity class (together with its subclasses, if any)

- Prefer this to EntityManager if (large) collections of objects...

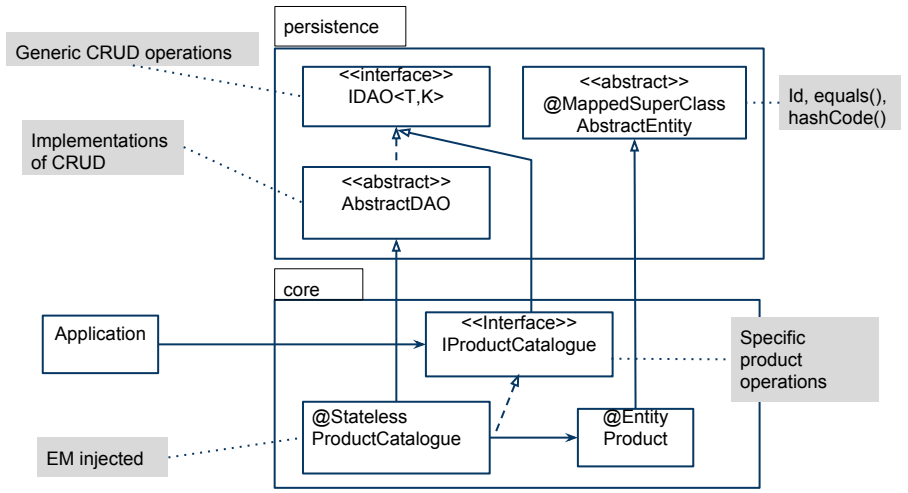
Persistence Layer Interface

```
// T entity type, K key type
public interface IDAO<T, K> {
    public void create(T t);
   public void delete(K id);
   public void update(T t);
   public T find(K id);
   public List<T> findAll();
   public List<T> findRange(int first, int n);
   public int count();
}


// Non-OO approach
Database db = Database.getInstance();
db.save( myOrder );  // Persist explicitly

// Much more OO
OrderBook ob = ....getOrderBook();
ob.add( myOrder );  // Implicit persist (transparent)
```

Basic interface for any class handling entities

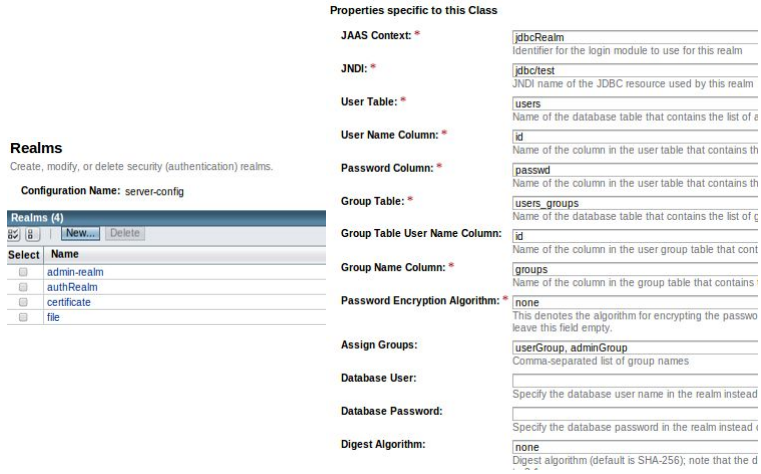
- AKA **Data Access Objects (DAOs)**
- Normally implemented by database aware collections
  - Using stateless EJBs
  - All JPA entity object handling in DAOs
- Transparent persistency, relevant model object handles persistence (nice)

Persistence Layer Design



See code sample jpa_persistence_layer

JEE Authority Revisited



Previously used a file realm for JEE authorization.

- Database (JDBC) realm better
- Must create a GlassFish (JDBC) realm to connect server and database (use Admin Console)
    - Must have database with table for users (some column must be specified as login and password columns and more...)
    - See code sample

More on Persistence layer design

- We possible look at a more complex sample