# Object-Relational Mapping

Jakub Daněk (KIV ZČU, Yoso Czech s.r.o.)

ORM Basics, JPA

http://blog.danekja.org/about (Twitter & LinkedIn links)
http://www.yoso.fi (yep, in Finnish)

# Motivation: ORM vs SQL

## JDBC example

```
Statement psmt = new PreparedStatement(“SELECT * FROM Users”);

ResultSet set = psmt.execute();

List<User> users = new ArrayList();
while(set.next()) {
    String name = set.get(“name”);
    String email = set.get(“email”);
    User u = new User(name, email);

    users.add(u);
}
```

## ORM example

```
List<User> users = repository.findAll(User.class);
```

# What is ORM?

- Bridge between object and relational worlds

- Commonly – each class represented by single table
  - With a few exceptions (inheritance, embedding)

- Boilerplate code existing in most projects
  - Classes to tables
  - Attributes to columns

- Mapping of queries to methods

- Developers don’t need to code SQL
  - But really?

# What ORM is NOT!

- Silver bullet
  - Always check your usecase
  - Be aware of weaknesses
- Perfect abstraction from relational database
  - Ignoring of relational concepts leads to serious problems
    - Performance issues
    - Software evolution complications
- High-performance solution
  - Mapping code overhead
  - Inefficient queries

# ORM Categories

- Fully Automated ORM
  - Automated SQL generation → developer writes none
  - Easy to “use”, hard to use “right”
  - Some sources claim only this is “ORM”
  - e.g. Hibernate (JPA)

- “Manual” ORM
  - Developer still writes SQL
  - Framework provides help with mapping
    - Rows to instances
    - Queries to methods

# Automatic ORM Example

## JPA Class mapping

```
@Entity
@Table(name=”app_user”)
class User {

    @Id
    @Columns(name=”id”)
    private Long id;

    @Column(name=”username”)
    private String username;

    @Columns(name=”email”)
    private String email;

    //getters, setters
    //equals, hashcode
}
```

## JPA Class save/read

```
User user = … //new user

user = repository.persist(user);

…

User u = repository.find(1L, User.class);
```

# Manual ORM Example

## MyBatis class example

```
public interface UserMapper {
 @Results({
   @Result(property = "id", column = "id"),
   @Result(property = "userName", column = "user_name"),
   @Result(property = "email", column = "email")
 })

 @Select("SELECT * FROM t_user")
 User selectUser(Long id);

 @Insert(“INSERT INTO t_user (user_name, email) VALUES (#{userName}, #{email})”)
 void insertUser(User u);
}
```

# Manual ORM Example

## MyBatis class example

```
User user = … //new user
UserMapper repository; //let’s not care about how to get the instance

repository.insertUser(user);

…

User u = repository.selectUser(1L);
```

# Issues – Impedance Mismatch

- **Object and Relational models are not equal**
  - ORM often prevents “perfect” object design
  - **Accessibility Control**
    - Relational database doesn’t have private fields
    - Commonly need to provide free access to attributes that are supposed to be private
      - In Java → getters and setters for everything

# Issues – Impedance Mismatch

- **Inheritance**
  - Relational database doesn’t know these terms
  - Several approaches to deal with inheritance:
    - **Table per class** – each class in the hierarchy has own table

# Issues – Impedance Mismatch

- **Inheritance**
  - Several approaches to deal with inheritance:
    - Table per class – each class in the hierarchy has own table
    - **Single table** – all classes in the hierarchy in single table
      - Lots of “blank” columns in each row

# Issues – Impedance Mismatch

- **Inheritance**
  - Several approaches to deal with inheritance:
    - Table per class – each class in the hierarchy has own table
    - Single table – all classes in the hierarchy in single table
      - Lots of “blank” columns in each row
    - **Joined tables** – one table for superclass, one table per subclass
      - Efficient data storage, but lots of joins (performance impact)

# Issues – Impedance Mismatch

- **Relations**
  - Object oriented concepts offer bigger variety
    - Association, composition, aggregation
  - In-memory access to associations is simpler than in relational database, compare:

    -
      ```
      User u = getCurrentUser();
      List<Role> roles = user.getRoles()
      ```

    -
      ```
      SELECT * FROM user WHERE id = 10;
      SELECT r.id, r.name FROM role r LEFT JOIN user_role ur
      ON r.id = ur.role_id WHERE ur.user_id = 10;
      ```

# Issues – performance

- **ORM brings performance issues**
  - Operational overhead
    - Logic for automatic query generation and attribute resolving is slow
      - requires reflection
  - Programmer must be aware that method calls result in SQL queries

# Issues – performance

- **Association Fetching**
  - Requires loading data from multiple tables

```
Class User {
    String name;
    Address address;
}
SELECT * FROM user WHERE;
//for all users
SELECT * FROM address a WHERE = a.user_id = :userId
```

# Issues – performance

- **N + 1 SELECT problem**
  - 1 query to list N items (users, forum topics…)
  - N queries to fetch additional data (address, comments)
  - Eventually may result in loading whole database in order to display single page
    - Hunders of database queries to display single page
      - Acceptable performance on dev machine (single user, little data), collapses in production

# Issues – performance

- **N + 1 SELECT problem – solution?**
  - LAZY loading of associations
    - Associations not fetched by default
    - Attribute is not filled with object until accessed (getter called)
      - Causes issues in some implementations (we shall see later on)
        - The getter is still there, even though the object is not filled

# Issues – performance

- **N + 1 SELECT problem – what if we need the data?**

  - For *-to-one associations

    - Use JOIN instead of multiple selects

      ```
      SELECT * FROM user WHERE;
      //for all users
      SELECT * FROM address a WHERE = a.user_id = :userId
      ```

      becomes

      ```
      SELECT u.username, a.city FROM user u LEFT JOIN address a ON
      u.id = a.user_id;
      ```

# Issues – performance

- **N + 1 SELECT problem – what if we need the data?**
  - For *-to-many associations
    - JOIN would result in increased results dataset with lot of duplicate data → mapping rows to objects takes time
      - Moves performance issue from database to ORM framework

# Issues – performance

- **N + 1 SELECT problem – what if we need the data?**
  - For *-to-many associations
    - Common usecase is have list of items (discussion topics) and on-click show details (comments) for a single one
      - Have separate method for loading the collection association only when needed → solves N+1 by limiting N to value 1

# Issues – performance

- **N + 1 SELECT problem – what if we need the data?**
  - For *-to-many associations
    - If you need to load collections for all returned items:
      - First load the main items (discussion threads) – **1 query**
      - Second load the associations (comments) in bulk for all – **1 query**
        - Assign items to their owners in-memory

# Issues – performance

- **General Rules**
  - THINK!
  - Always map *-to-many associations (collections) as LAZY
  - Always load non-lazy (EAGER) *-to-one associations using JOIN
  - Do not pretend the relational database isn't there

# Issues – Leaky Abstraction

- **Results**

  - ORM influences how you design your application

    - Data structure – object interface and attribute accessibility

    - Application interface design

→ **leaky abstraction, but there is not much we can do about it**

# ORM Implementations

- JAVA
  - Hibernate, EclipseLink, OpenJPA, MyBatis
- Ruby
  - ActiveRecord, DataMapper, Sequel
- Python
  - Django’s ORM, SQLAlchemy, Peewee, SQLObject
- .NET
  - Entity Framework, nHibernate

# Java Persistence API (JPA)

# What is JPA (1)

- Standarized ORM **interface** for Java
  - Implementations: Hibernate, EclipseLink, OpenJPA

- Part of JavaEE specification

- Current version 2.1

# What is JPA (2)

- Entity Metadata
  - **Annotations**, XML

- Java Persistence Query Language (**JPQL**), SQL-like
    - Classes vs Tables
    - Attributes vs Columns
    - Associations vs Relations

- Query API - “Criteria API”
  - Programatical querying API

# JPA – Basic Terms (1)

**JPA Entity**

- Instance managed by the persistence framework
- Non-transient fields persisted to data store
- Has own lifecycle

# JPA – Basic Terms (2)

**Persistence Unit**

- Set of entity **types** managed by the persistence framework
- Entity **classes**, instances of which are persisted into the same data store by the application

# JPA – Basic Terms (3)

**Persistence Context**

- Set of entity **instances** managed by the persistence framework
- For each PK in the store there is a unique entity instance
- Basically a **cache** representing the stored data
- Scope: typically a **transaction**

# JPA – Basic Terms (4)

**Entity Manager**

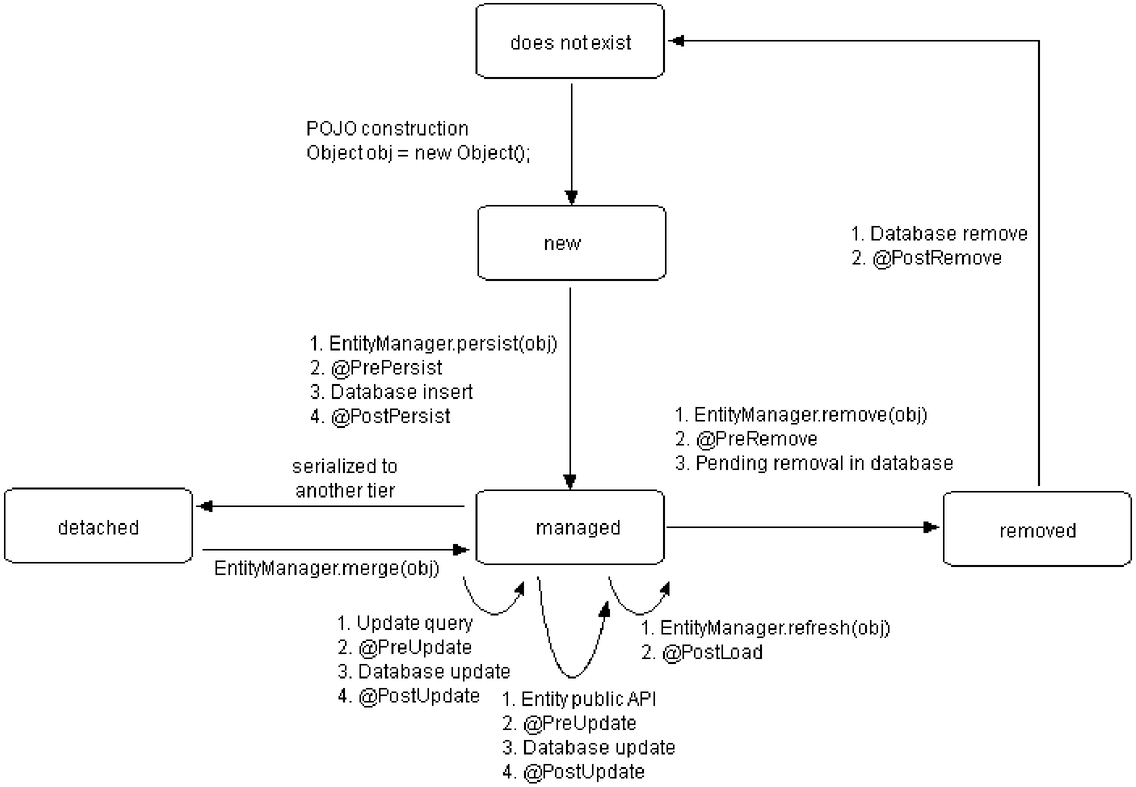
- Object used to manage entity instances in **persistence context**
- API to create, update, remove entity instances
- API to query over entity instances
- Basically a **DAO**

# JPA – Basic Terms (5)

**Entity Manager Factory**

- Object used to create **Entity Manager**
- API to create, update, remove entity instances
- API to query over entity instances

# JPA – Entity Lifecycle



does not exist

POJO construction
Object obj = new Object();

new

1. EntityManager.persist(obj)
2. @PrePersist
3. Database insert
4. @PostPersist

1. Database remove
2. @PostRemove

1. EntityManager.remove(obj)
2. @PreRemove
3. Pending removal in database

serialized to
another tier

detached

managed

removed

EntityManager.merge(obj)

1. Update query
2. @PreUpdate
3. Database update
4. @PostUpdate

1. Entity public API
2. @PreUpdate
3. Database update
4. @PostUpdate

1. EntityManager.refresh(obj)
2. @PostLoad

Src: https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm#CIHCJGGJ

# JPA – Entity Mapping (class mapping)

- **Entity** has to:
  - Be a **JavaBean**
  - Be annotated with **@Entity**
  - Have primary key attribute annotated with **@Id**

- Custom table name - **@Table** (optional)

```
@Entity
@Table(name=”app_user”)
public class User {
    @Id
    public Long getId(){};
}
```

# JPA – Entity Mapping (attributes)

- All bean attributes are persisted by default
  - Explicit annotations are recommended, but not needed
- Annotations
  - @Basic – elementary datatypes (optional)
  - @Temporal – date-related datatypes (Date, Instance)
  - @Enumerated – enums
- What if I don’t want to persist particular attribute?
  - @Transient – marks attributes that shouldn’t be mapped to database

# JPA – Entity Mapping (attributes)

```
@Entity
public class User {
   @Basic
   String getUsername(){};

   @Temporal()
   Date getDateOfBirth(){};

   @Enumerated
   UserState getState(){};

   @Transient
   int getAge() {
      //get age from date
   }
}
```

```
public enum UserState {
   NEW,
   ACTIVE,
   DELETED;
}
```

# JPA – Entity Mapping (attributes)

- Column modification
  - **@Column** – allows for better specification of column
    - Column name
    - Bool flags – nullable, insertable, updatable
    - Default value etc.
    - Can be used in conjuction with other annotations

```
@Basic
@Column(name=”user_name”, updatable=false)
public getUsername() {};
```

# JPA – Entity Mapping (associations)

- Assocations between **entities**
  - **1..1, 1..N, N..1, M..N**

- Annotations
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany

- One entity is always the **owner** of the association
  - e.g. **User** has **Address** → User is owner of the association

# JPA – Entity Mapping (associations)

**User** has **Address –** unidirectional mapping

```
@Entity
public class User {

@ManyToOne
Address getAddress() {};

}
```

```
@Entity
public class Address {

      ...

}
```

# JPA – Entity Mapping (associations)

- Sometimes it is suitable to have attributes on both sides of single association → **bidirectional mapping**

- **User** is still **owner** of the association

```
@Entity
public class User {

@ManyToOne
Address getAddress() {};

}
```

```
@Entity
public class Address {

@OneToMany(mappedBy=”address”)
List<User> getUser() {};

}
```

# JPA – Entity Mapping (associations)

- Double unidirectional (without **mappedBy**) vs bidirectional mapping
  - Double unidirectional – two independent associations
  - Bidirectional – single association described on both ends

- Mapping a bidirectional association as unidirectional gives wrong results
  - Counterpart value is not set properly by the persistence framework

# JPA – Entity Mapping (associations)

**Unidirectional Mapping**

```
@Entity
public class User {

@ManyToOne
Address getAddress() {};

}
```

```
@Entity
public class Address {

@OneToMany
List<User> getUsers(){};

}
```

# JPA – Entity Mapping (associations)

**Unidirectional Mapping**

```
EntityManager em;

User u = em.find(User.class, 1L);
Address a = em.find(Address.class, 100L);

u.setAddress(a);
em.update(u);
//User with id 1 now owns Address with ID 100

a = em.find(Address.class, 100L);
a.getUsers().isEmpty() //true
```

# JPA – Entity Mapping (associations)

**Bidirectional Mapping**

```
@Entity
public class User {

@ManyToOne
Address getAddress() {};

}
```

```
@Entity
public class Address {

@OneToMany(mappedBy=”address”)
List<User> getUsers(){};

}
```

# JPA – Entity Mapping (associations)

**Bidirectional Mapping**

```
EntityManager em;

User u = em.find(User.class, 1L);
Address a = em.find(Address.class, 100L);

u.setAddress(a);
em.update(u);
//User with id 1 now owns Address with ID 100

a = em.find(Address.class, 100L);
a.getUsers().isEmpty(); //false
a.getUsers().contains(u); //true
```

# JPA – Entity Mapping (associations)

- Association loading – LAZY vs. EAGER

- EAGER
  - Association fetched with the owning entity
  - Usually not used – performance issues (remember N+1 SELECT?)
  - If used, needs to be optimized on query level (we shall see further)

# JPA – Entity Mapping (associations)

- Association loading – LAZY vs. EAGER

- LAZY
  - Association fetched on attribute access (getter call)
  - Causes a lot of pain to programmer, still it is a necessity
  - Owning entity must be in **managed** (not **detached!**) state
    - Otherwise results in **LazyInitializationException**
    - Often not the case (e.g. in UI) → getters sometimes work, sometimes not (remember **leaky abstraction**?)

# JPA – Entity Mapping (associations)

- Association loading – so how?
- Set-up strict project rules (1)
  - **Avoid using getters and setters for collections where possible**
    - At least above your transaction level – typically anything above business logic level
    - Depends on your usecase
      - e.g. when loading user, you probably quite often need his roles in the same view (page)
      - And commonly you load only single (current) user object

# JPA – Entity Mapping (associations)

- Association loading – so how?
- Set-up strict project rules (2)
  - **Use custom DAO methods for fetching collections**
    - Typically you display details on a separate page
      - Example: Forum – you don’t need to see all Thread posts on “Topic Listing” view
    - Result: constant number of queries required to display a page
      - **Eliminates N+1 problem**

# JPA – Entity Mapping (associations)

- Association loading – so how?
- Set-up strict project rules (2)
  - **Use custom DAO methods for fetching collections**

```
//on Topic Listing
List<Topic> topics  = topicDao.findAll();


//on Topic click, single thread opens
List<Post> posts = postDao.findByTopic(currentTopic.getId());
```

# JPA – Entity Mapping (associations)

- Association loading – so how?
- Set-up strict project rules (3)
  - **Optimize ALL queries for fetching 1..1 and M..1 associations**
    - Enforce using JOIN for fetching *ToOne associations → only one select query per entity instance
      - HOW: JPQL and Criteria API later on
      - **Eliminates N+1 problem**
    - No way to make this default :(

# JPA – Entity Mapping (associations)

- Association loading – so how?
- Set-up strict project rules (3)
  - **Optimize ALL queries for fetching 1..1 and M..1 associations**
    - Default:

```
SELECT username, address_id FROM app_user WHERE id = 1;

//take address_id and use it in next query
SELECT * FROM address WHERE id = :address_id
```

    - Better:

```
SELECT * FROM app_user u LEFT JOIN address a ON
u.address_id = a.id WHERE u.id = 1;
```

# JPA – Entity Mapping (associations)

- Association loading – so how?
- Set-up strict project rules (Question)
  - **Optimize ALL queries for fetching 1..1 and M..1 associations**
    - **Question:** Why not use it for collection fetching as well?

# JPA – Entity Mapping (associations)

- Association loading – so how?
- Set-up strict project rules (Question)
  - **Optimize ALL queries for fetching 1..1 and M..1 associations**
    - Question: Why not use it for collection fetching as well?
    - **Answer:** Because the query would return duplicate data for the owning entity (one line per collection item)
      - **Framework handles it (we don’t get duplicate results), but it has performance impact**

# JPA – Entity Mapping (associations)

- **Embedding**
  - “embedded” entity stored in the owners table.

```
@Entity
Class User {
    @Id
    Long getId() {};

    @Embedded
    Address getAddress() {}

    String getUsername(){};
}
```

```
@Embeddable
class Address {
    String getStreetName() {};
    String getCity() {};
}
```

- Table User has columns: id, username, **streetName, city**

# JPQL and Criteria API

# JPQL - Java Persistence Query Language

- Java Persistence Query Language (JPQL)
  - Standarized query language for JPA, inspired by SQL
  - Implementations have own mutations
    - Avoid if possible
    - Hibernate: Hibernate Query Language (HQL)
    - EclipseLink: EclipseLink Query Language (EQL)

  - Reference: http://www.objectdb.com/java/jpa/query

# JPQL - Java Persistence Query Language

- Main differences from SQL
  - Uses entity class names instead of tables
  - Uses attributes instead of column names
    - It is possible to traverse attribute path (user.address.streetName)
- One JPQL query may map to several SQL queries
  - Association fetching

# JPQL – Basic Example

```
package org.danekja;

@Entity
@Table(name=”app_user”)
public class User {

    @Column(name=”user_name”)
    public String getUsername() {};

}
```

- **SQL:**

```
SELECT * FROM app_user u WHERE u.user_name = “Karel”;
```

- **JPQL:**

```
SELECT u FROM org.danekja.User u WHERE u.username = “Karel”
```

# JPQL – LEFT JOIN

```
package org.danekja;

@Entity
@Table(name=”app_user”)
public class User {

@OneToMany
List<Address> getAddresses() {};

}
```

```
package org.danekja;

@Entity
@Table(name=”address”)
public class Address {

@Column(name=”street_name”)
String getStreetName()

}
```

- **SQL:**

```
SELECT u.id, u.username FROM app_user u LEFT JOIN address a ON
u.id = a.user_id WHERE a.street_name = “Technicka”;
```

- **JPQL:**

```
SELECT u FROM org.danekja.User u LEFT JOIN u.adresses a WHERE
a.streetName = “Technicka”
```

# JPQL – LEFT JOIN

**Note:** You cannot do LEFT JOIN on two entities that don’t have association mapped

i.e. the following **doesn’t** work: `FROM User u LEFT JOIN Address a` (!!!)

- **SQL:**

```
SELECT u.id, u.username FROM app_user u LEFT JOIN address a ON
u.id = a.user_id WHERE a.street_name = “Technicka”;
```

- **JPQL:**

```
SELECT u FROM org.danekja.User u LEFT JOIN u.adresses a WHERE
a.streetName = “Technicka”
```

# JPQL – JOIN FETCH

- Fetching *-to-one associations
  - Default is separate SELECT (causes N+1 issue)
  - How to enforce fetching *-to-one association using JOIN?

- **SQL:**

```
SELECT * FROM app_user u LEFT JOIN address a
          ON u.address_id = a.id WHERE u.id = 1;
```

- **JPQL:**

```
SELECT u FROM User u JOIN FETCH u.address WHERE u.id = 1;
```

# JPQL – Executing Query

- Dynamic queries
  - Translated to SQL at runtime
  - Performance impact – implementations try to cache
- Named queries
  - Translated to SQL at startup
  - Have unique name

# JPQL – Dynamic Query

Typed:

```
TypedQuery<Country> query =
      em.createQuery("SELECT u FROM User u", User.class);

List<User> results = query.getResultList();
```

Or untyped:

```
Query query = em.createQuery("SELECT u FROM User u");
List<Object> results = query.getResultList();
```

# JPQL – Named Query

- Annotations **@NamedQueries** and **@NamedQuery**

```
@Entity
@NamedQueries({
   @NamedQuery(name=”User.findByUsername”,
               query=”SELECT u FROM User u
                              WHERE u.username = :name)
})
public class User {
   public String getUsername() {};
}
--------------------
TypedQuery<User> q =
   em.createNamedQuery(“User.findByUsername”, User.class);

List<User> results = q.getResultList();
```

# JPQL – Query Parameters

- Ordinal parameters
  - Format: ?index – e.g. ?3

```
Query q = em.createQuery(“SELECT u FROM org.danekja.User u LEFT
JOIN u.adresses a WHERE a.streetName = ?1”);

q.setParameter(1, “Technicka”);

List results = q.getResultList();
```

# JPQL – Query Parameters

- Named parameters
  - Prefixed by ‘**:**’ - e.g. **:name**
  - Preferred

```
Query q = em.createQuery(“SELECT u FROM org.danekja.User u LEFT
JOIN u.adresses a WHERE a.streetName = :streetName”);

q.setParameter(“streetName”, “Technicka”);

List results = q.getResultList();
```

# Criteria API

- Programmatical API for building queries
- Same power as JPQL queries
- More suitable for building dynamic queries at runtime
  - e.g. when there is a lot of optional fields
  - Avoids String concatenation

- Reference: http://www.objectdb.com/java/jpa/query/criteria

# Criteria API – Basic Example

- **JPQL:**

```
SELECT u FROM org.danekja.User u WHERE u.username = “Karel”
```

- **Criteria API:**

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<User> query = cb.createQuery(User.class);

//FROM
Root<User> root = query.from(User.class);

//SELECT, WHERE
q.select(root).where(cb.equal(root.get(“username”), “Karel”);
```

# Criteria API – Basic Example

- **Running a Criteria API query:**

```
CriteriaQuery<User> q;
q.select(root).where(cb.equal(root.get(“username”), “Karel”);

//create a TypedQuery based on criteria query
//just like for JPQL queries
TypedQuery<User> tq = em.createQuery(q);

ResultList<User> results = tq.getResultList();
```

- The **CriteriaQuery** is equivalent to a JPQL string
  - that’s why we need to create a **TypedQuery** for execution

# Criteria API – Query Building

- Where clause logical join

```
//JPQL
(u.firstName = “Karel” OR u.alias = “Carlos”) AND u.lastName = “Novák”

//Criteria API
Root<User> u = query.from(User.class);

Predicate fn = cb.equal(u.get(“firstName”), “Karel”);
Predicate al = cb.equal(u.get(“alias”), “Carlos”);
Predicate ln = cb.equal(u.get(“lastName”), “Novák”);

Predicate or = cb.or(fn, al);
Predicate and = cb.and(or, ln);

query.where(and);

...
```

# Criteria API – Query Building

- JOIN (1)

```
//JPQL
SELECT u1, u2 FROM User u1, User u2

//Criteria API
Root<User> u1 = query.from(User.class);
Root<User> u2 = query.from(User.class);

query.multiselect(u1, u2);

...
```

# Criteria API – Query Building

- JOIN (2)

```
//JPQL
SELECT u, a FROM User u LEFT JOIN u.address

//Criteria API
Root<User> u = query.from(User.class);
Join<User> a = u.join(“address”, JoinType.LEFT);

query.multiselect(u, a);

...
```

# Criteria API – Query Building

- JOIN (3)

```
//JPQL
SELECT u FROM User u JOIN FETCH u.address

//Criteria API
Root<User> u = query.from(User.class);
Fetch<User, Address> a = u.fetch(“address”);

query.select(u);

...
```

# Queries - Final Thoughts

- Use both JPQL and Criteria API
- Think about the query effectivity
- Be careful about the N+1 select problem

# Sources

- https://blog.jooq.org/2015/08/26/there-is-no-such-thing-as-object-relational-impedance-mismatch/
- http://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html
- http://tomee.apache.org/jpa-concepts.html
- http://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html
- https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm#CIHCJGGJ
- http://www.objectdb.com//java/jpa

Thank You!