Java Persistence API

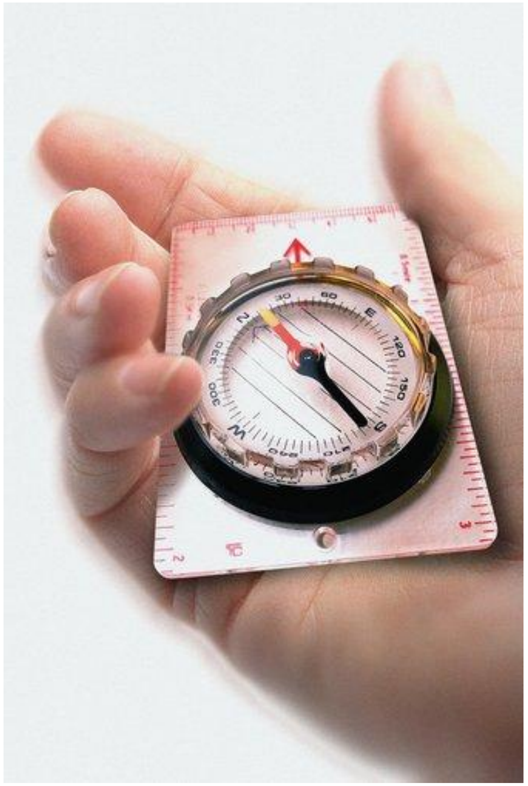
Object Relationnal Mapping

Course objectives

By completing this course you will be able to:

- Explain what is JPA
- Use Java Persistence API to persist data
- Use some famous persistence layer patterns

Course plan



- JPA Entity
- JPA – Advanced functions
- JPQL
- Good practices

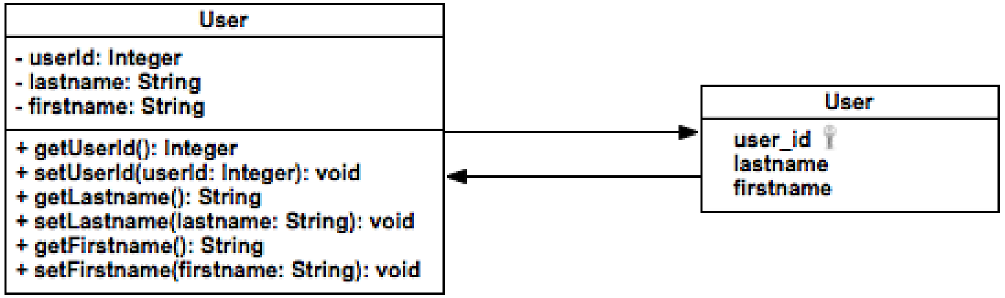
Java Persistence API

JPA ENTITY

Or how to manage our database in a transparent way

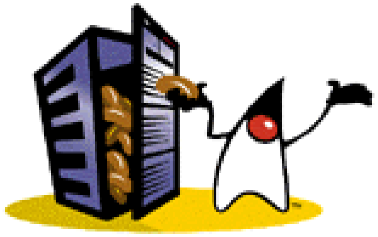
Overview

- JPA is part of the Java EE platform
  - Inspired from Frameworks like Hibernate and JDO
  - Relies heavily on annotation feature
- Relationship between objects and tables is done automatically (ORM: “object-relational” mapping)



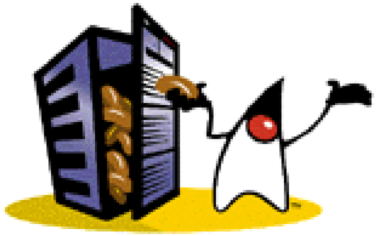
Overview

- Necessary items:
  - A relational database
  - A JDBC driver as a jar
  - A XML configuration file for database access
  - A JavaBean class, which will become a JPA Entity with some annotations
  - A JPA Entity Manager



Overview

- Necessary items:
  - A relational database
  - A JDBC driver as a jar
  - A XML configuration file for database access
  - A JavaBean class, which will become a JPA Entity with some annotations
  - A JPA Entity Manager



Relational database

- The majority of relational databases
  - MySQL
  - PostGreSQL
  - Oracle
  - SQL Server
  - DB2
  - …

Entity annotation

- A JPA Entity is just a POJO with private properties, getters and setters, default constructor :

```
public class Contact implements Serializable {
    // my properties
    private int id;
    private String name;
    private String firstname;

    // ... setters and getters ...
}
```

JPA Entity

Entity annotation

- Embellished with annotations:

```
@Entity
public class Contact implements Serializable {
    // my properties
    @Id
    private int id;
    private String name;
    private String firstname;

    // ... setters and getters ...
}
```

Entity annotation

- @Entity annotation is put on the class : it is used to declare a class as a “JPA Entity”
- @Table annotation allows to define the name of the table to which the class is mapped (optional)

```
@Entity
@Table(name="CONTACTS")
public class Contact  implements Serializable {

}
```

Properties annotation

- **@Id** annotation is set on the property or on the getter representing the primary key within the database
- It's possible to define how to generate the key with the annotation :
  - **@GeneratedValue**(strategy=GenerationType.XXX)
- The constants proposed are :
  - IDENTITY - TABLE
  - SEQUENCE - AUTO

Properties annotation

- Some important annotations:

Annotation	Description
@Basic	If no specific annotation is declared, this one is used
@Transient	When we do not want to make a property persistent
@Lob	Allows to stock big strings, byte arrays, …
@Temporal	Used to persist dates, hours
@Enumerated	Specify an enumerated field

Persistence providers

- There are different JPA implementations:

HIBERNATE

ORACLE TOPLINK

eclipse link

OpenJPA

- The code remains the same, only the configuration file changes

Persistence unit

- The persistence unit is the key element of the JPA Entity technology
- It “persists” entities in the database.
- Requires a persistence provider and other configurations inside a special file : persistence.xml

persistence.xml Example 1/2

```
<?xml version="1.0"?>
<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    version="2.0">

    <persistence-unit name="My-PU"
        transaction-type="RESOURCE_LOCAL">

        <provider>
            org.hibernate.ejb.HibernatePersistence
        </provider>

        <properties>
            <property
              name="javax.persistence.jdbc.driver"
              value="com.mysql.jdbc.Driver" />
...
```

persistence.xml Example 2/2

```
...
              <property
                 name="javax.persistence.jdbc.user"
                 value="root" />
              <property
                 name="javax.persistence.jdbc.password"
                 value="root" />
              <property
                 name="javax.persistence.jdbc.url"
                 value="jdbc:mysql://host:3306/MyDB" />
             <property
                   name="hibernate.hbm2ddl.auto"
                   value="update" />
         </properties>
    </persistence-unit>
</persistence>
```

JDBC driver

- Each database provides a JDBC driver to access it through Java
- Depending on the database used, the appropriate JAR should be put in the libraries

Entity Manager

- As its name implies, the Entity Manager object will handle all operations on entities : inserting, modifying, deleting them in the database
- No SQL code is required, we manipulate Java objects directly :

```
Country c = new Country("France");
EntityManager em = ...
em.persist(c);
em.close();
```

Entity Manager

- Some common operations :
  - void persist(Object entity)
  - <T> T merge(T entity)
  - void remove(Object entity)
  - <T> T find(Class<T> entityClass, Object primaryKey)

- Thanks to them, there is almost no request to write

EntityManager

- How to retrieve it ?
  - Use an EntityManagerFactory !

```
EntityManagerFactory emf = null;
emf = Persistence.createEntityManagerFactory("My-PU");
EntityManager em = emf.createEntityManager();
Contact contact = em.find(Contact.class, 1);
em.close();
emf.close();
```

EntityManager

- Before writing in DB, you should begin a transaction!

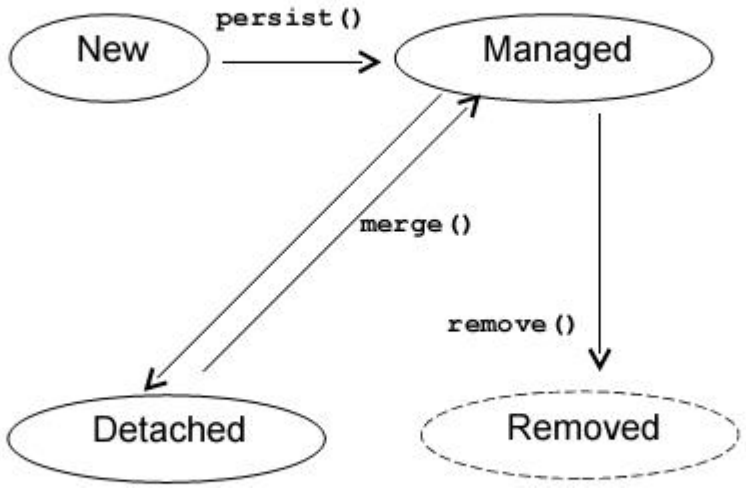
```
EntityManager em = emf.createEntityManager();
EntityTransaction t = em.getTransaction();
try {
    t.begin();
    // ...
    t.commit();
} finally {
    if (t.isActive()) t.rollback();
    em.close();
}
```

EntityManager

- EntityManager objects are not thread safe…
  - Don’t define one as servlet instance variable !
- … but EntityManagerFactory is.
  - You can use the same instance for all your application
- Think to close your EntityManager and EntityManagerFactory object !

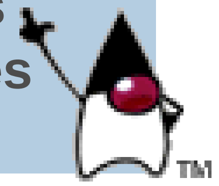
Entity states

- **Transient**: After the calling of key **new**
- **Managed**: after the calling of method **persist()**
- **Detached**: when the object is manipulated on the client
- **Removed**: object removed from database

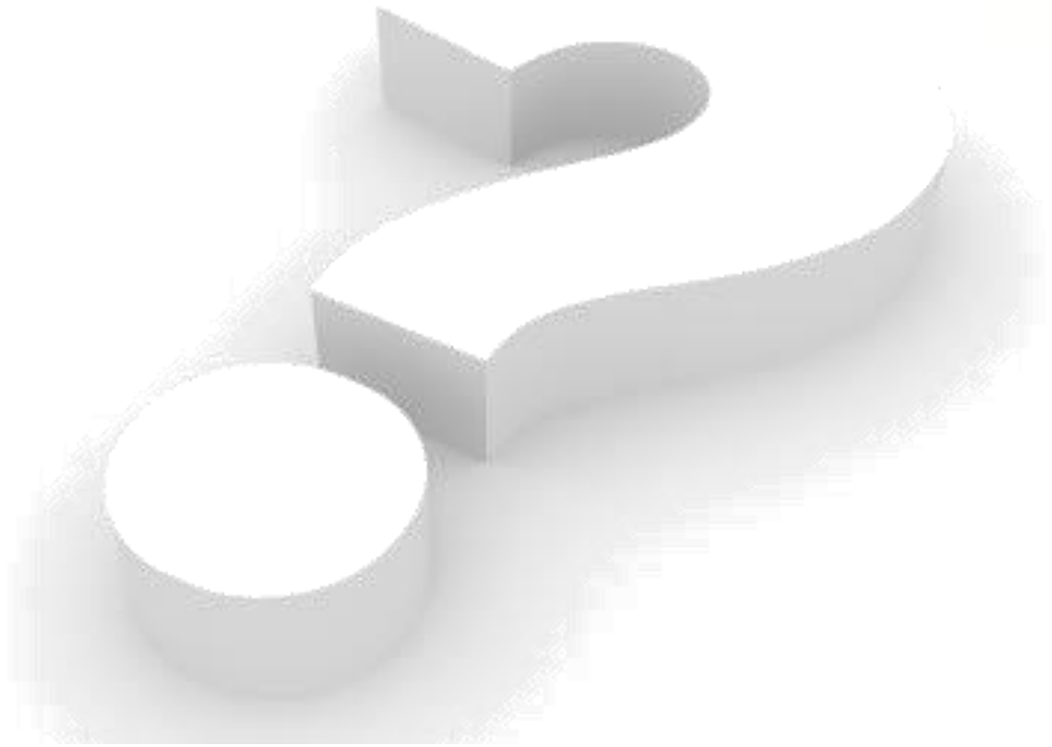


Quizz

Which annotation allows to declare a JPA Entity ?	@Entity
Which annotation allows to declare a primary key?	@Id
How to declare the connection with a Database ?	By deploying a file describing a persistence unit
What is the use of Entity Manager ?	Its methods handles persistence of Entities



Questions ?



Exercises (1/3)

- Add Hibernate libraries to your project
- Add the MySQL JDBC library too
- Create a JavaBean class named **Category**
  - In a package **com.supinfo.supcommerce.entity**
  - With **id** as Long and **name** as String
- Transform it into a JPA Entity
  - The table should be named **categories**
  - The **id** field must be the primary key of the table

Exercises (2/3)

- Create a PersistenceUnit
- Create a JSP page named **addCategory.jsp**
- Create an **HttpServlet** named **AddCategoryServlet**
  - Bind it to **/auth/addCategory** url-pattern
  - Override the **init()** method
    - Create an EntityManagerFactory object
  - Override the **destroy()** method
    - Close the EntityManagerFactory object

Exercises (3/3)

- Create an **HttpServlet** named **AddCategoryServlet**
  - Override the **doPost()** method
    - Retrieve the form parameters
    - Create a new **Category** object
    - Set the parameters in the object
    - Use an EntityManager to persist the object
  - Override the **doGet()** method
    - Forward the request to the JSP page

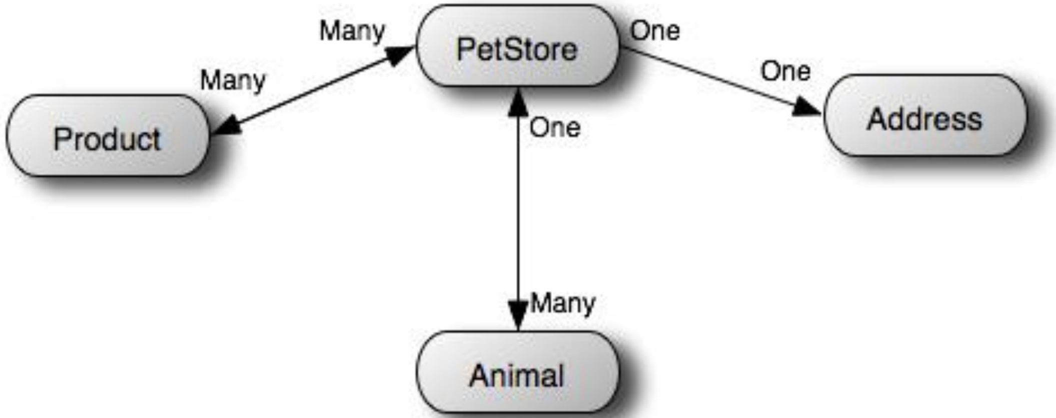
Java Persistence API

JPA – ADVANCED FUNCTIONS

Entities dependencies, inheritance

Relationship between entities

- Entities often have relationships between them :
  - One-To-One
  - One-To-Many
  - Many-To-One
  - Many-To-Many

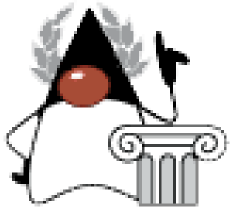


Relationship between Entity Beans

- Relations between entities are described with annotations put on the property or on the getter
  - Different strategies are available (foreign keys, join tables)
- JPA also handles inheritance relationship between entities with annotations

One-To-One

- @OneToOne annotation describes a one-to-one relation between two entities.
- There are 3 different strategies :
  - @JoinColumn
    - a foreign key is used
  - @PrimaryKeyJoinColumn
    - 2 dependent entities have the same primary key
  - @JoinTable
    - a join table contains primary keys



One-To-One

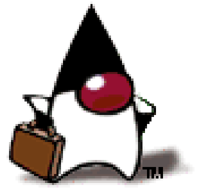
- For example, a store entity only has one address :

```
public class PetStore {
    ...
    @OneToOne
    @JoinColumn(name="address_fk")
    private Address address;
    ...
}
```

- In the pet store table, a foreign key is used

One-To-One and Many-To-One

- @OneToMany and @ManyToOne annotation link an entity to a collection of another entity
  - Example:
    - A person has several bank accounts, and each account has a unique owner
- Represented either by a join table or by a column as a foreign key
  - @JoinTable
  - @JoinColumn



One-To-One and Many-To-One

- Code example with a pet store selling many animals :

Store Entity	`@OneToMany(mappedBy="petStore")`
	`private Collection<Animal> animals;`

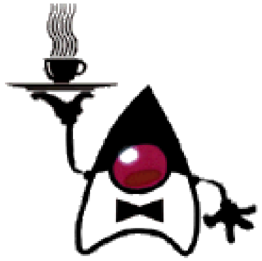
Animal Entity	`@ManyToOne`
	`@JoinColumn(name="store_fk")`
	`private PetStore petStore;`

- A foreign key column is added in the animal table

Many-ToMany

- @ManyToMany annotation binds 2 entities with each other
  - Example :
    - A product could have many categories and a category contains many products

- @JoinTable is the only option



Many-To-Many

- How to annotate your entities :

Store Entity	`@ManyToMany`
	 `@JoinTable(name="STORE_PRODUCT")`
	 `private Collection<Product> products;`

Product Entity	`@ManyToMany(mappedBy="products")`
	 `private Collection<PetStore> stores;`

- A join table represents the relationship between stores and products

Cascading

- All previous relationship annotations possess the cascade attribute
- An operation applied to an entity is reflected to dependent entities
  - Example : when a user is persisted, so is its accounts.
- Four types:
  - `PERSIST|MERGE|REMOVE|REFRESH`
  - `CascadeType.ALL` : 4 combined

Cascading

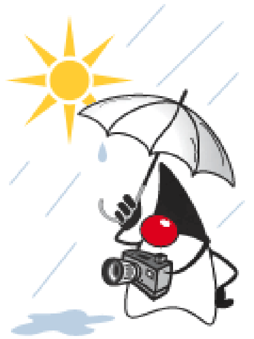
- The cascade attribute is set next to the annotation
  - PetStore entity with an unique address

PetStore Entity	`@OneToOne(cascade=CascadeType.PERSIST)`
	`@JoinColumn(name="address_fk")`
	`private Address address;`

- If the address doesn't exist in database, it is persisted at the same time as the store

Lazy loading

- All previous relationship annotations possess the fetch attribute
- When you retrieve an entity, multi-valued properties are not loaded by default.
  - Example : when a user is loaded, its accounts are not retrieved.
- 2 Types : `LAZY | EAGER`



Lazy loading

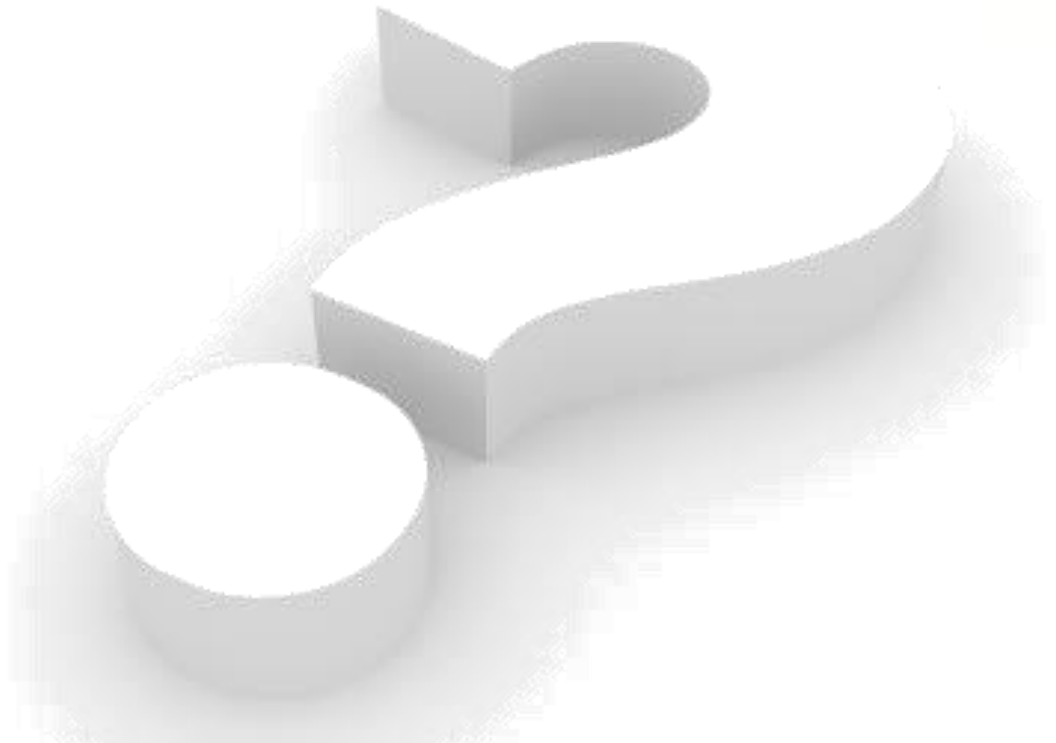
- By default, the *"lazy"* mode is applied for multi-valued properties (List, Set, Map, …)
  - Change it by putting the fetch property to *"eager"* on the annotation

PetStore Entity	`@OneToMany(mappedBy="petStore", fetch=FetchType.EAGER) private Collection<Animal> animals;`

```java
@OneToMany(mappedBy="petStore",
    fetch=FetchType.EAGER)
private Collection<Animal> animals;
```

- When a pet store is retrieved from database, its collection is initialized

Questions ?



Exercises (1/2)

- Create a JavaBean class named **Product**
  - In the package **com.supinfo.sun.supcommerce.entity**
  - With the same attributes than **SupProduct** class
- Transform it into a JPA Entity
  - The table should be named **products**
  - The id field must be the primary key of the table

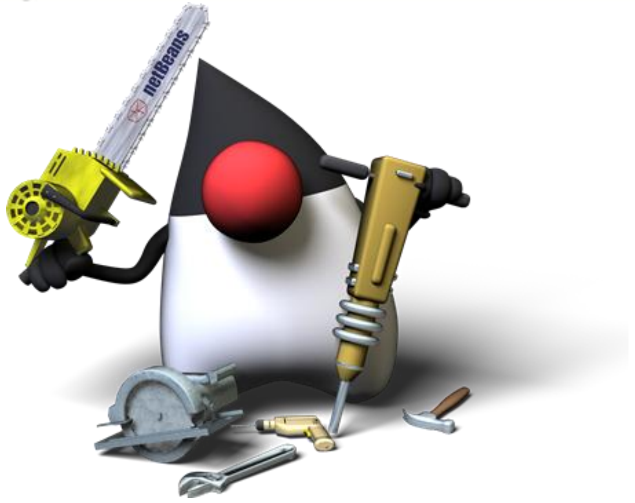
Exercises (2/2)

- Define a relationship between Product and Category entities
  - A product can only have one category
  - A category can have several products

- Update the **InsertSomeProductServlet**
  - Replace the **SupProduct** object by a **Product** one
  - Use EntityManager instead of **SupProductDao** class

Java Persistence API

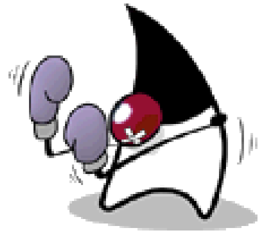
JPQL



The “object powered” SQL

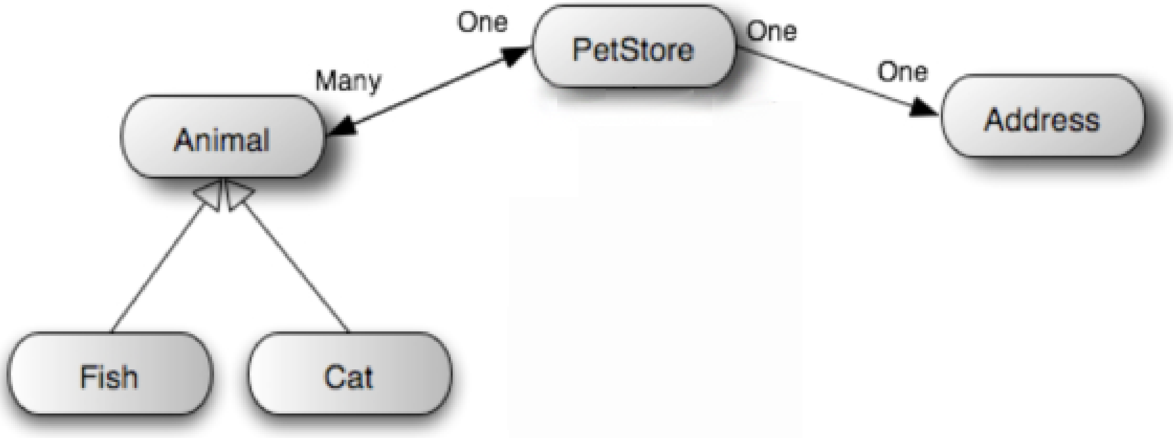
Presentation

- The Entity Manager handles basic CRUD* operations
- Java Persistence Query Language makes queries against entities stored in a relational database
- It looks a lot like SQL, many requests are available
- Instead of working on database tables, it manipulates Java objects transparently



Presentation

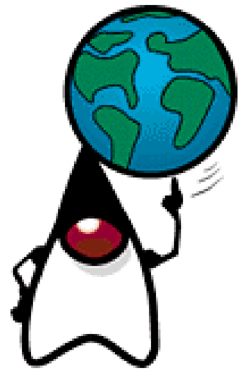
- JPQL manipulates the objects through an internal representation in the Entity Beans container



- It's called "abstract schema"

How-To

- To write a request, we need :
  - An Entity Manager
  - The JPQL language
  - A Query object
- The Entity Manager is able to create Query objects
- The Query is then executed



SELECT statement

- Reclaims all entries from an entity table
  - Obtain an Entity Manager
  - Create a Query object and then execute it

```
Query query = em.createQuery("SELECT c FROM Cat AS c");

List<Cat>list = query.getResultList();
```

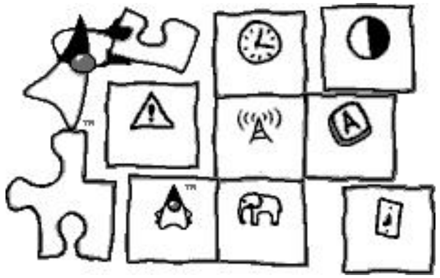
WHERE clause

- Apply conditions on a request

```
Query  query = em.createQuery("SELECT cat FROM Cat AS cat
   WHERE cat.animalId = 5");

Cat myCat = (Cat)query.getSingleResult();
```

- Some functions
  - BETWEEN
  - IS NULL
  - LIKE
  - …
- Order results with ORDER BY



DELETE and UPDATE statements

- Delete entities using JPQL

```
Query  query = em.createQuery("DELETE FROM Cat AS cat WHERE
   cat.earLength = 2");

int nbrDeleted = query.executeUpdate();
```

- Update entities using JPQL

```
Query  query = em.createQuery("UPDATE Cat AS cat SET
   cat.earLength = 3 WHERE cat.earLength = 4");

int nbrUpdated = query.executeUpdate();
```

Queries with parameters

- Parameters can be placed in queries
  - Numeric parameter

```
Query query = em.createQuery("SELECT cat FROM Cat AS cat WHERE
    cat.animalId = ?1");
query.setParameter(1, 5);
Cat myCat = (Cat)query.getSingleResult();
```

  - String parameter

```
Query query = em.createQuery("SELECT cat FROM Cat AS cat WHERE
    cat.animalId = :id");
query.setParameter("id", 5);
Cat myCat = (Cat)query.getSingleResult();
```

Aggregation functions

- Aggregation functions can be used with the SELECT clause
  - MIN
  - AVG
  - COUNT
  - SUM
  - …

```
Query query = em.createQuery("SELECT MAX(cat.earLength) FROM
   Cat AS cat");

Number maxEarLength = (Number) query.getSingleResult();
```

JPQL

Aggregation functions

- A special operator allows queries to work trough relationships : IN
- Example :
  - I want to get stores containing the product named "Product" :

```
Query query = em.createQuery("SELECT s FROM Store AS s,
   IN(s.products) AS p WHERE p.name = 'Product'");

List<Store> stores = (List<Store>) query.getResultList();
```

Named queries

- It's possible to declare named queries on the entity class
  - They're precompiled at deployment

```
@Entity
@NamedQuery(name="listBeverages", query="SELECT beverage FROM
   Beverage AS beverage")
public class Beverage implements Serializable{ … }
```

- How to call them

```
Query query = em.createNamedQuery("listBeverages");
```

Fill in the blanks

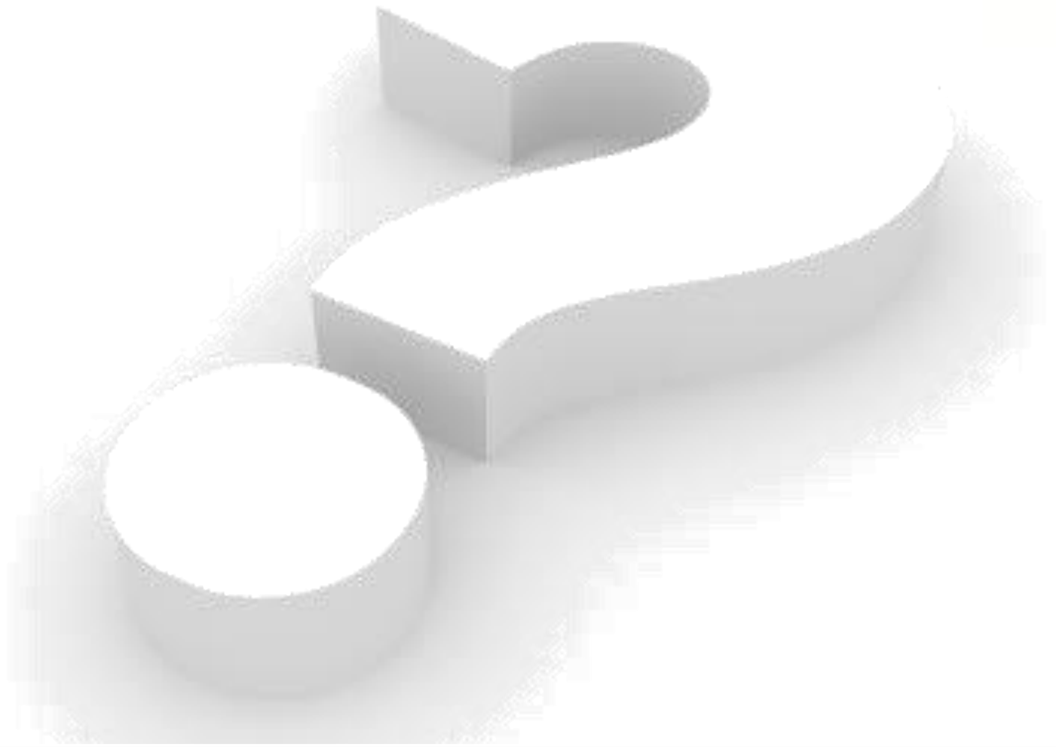
JPQL is language close from ….SQL…..

The interest is to manipulate the objects rather than
tables…………

The manipulation of requests is done with the class.Query….

The majority of SQL functions are still the same.

Questions ?



Exercises (1/3)

- Replace all your SupProduct objects by Product ones
- Use EntityManager instead of SupProductDao class
  - Think to close them!

- Create a **HttpServlet** named **CheaperProductsServlet**
  - Override the **doGet()** method
    - Retrieve all the products with price < 100 – **Use JPQL!**
    - Add them as request attributes
    - Forward the request to listProduct.jsp

Exercises (2/3)

- Update the **AddProductServlet**
  - In the **doGet()** method
    - Retrieve all categories and put them in request attribute
  - In the **doPost()** method
    - Retrieve the category id in request parameters
    - Retrieve with it the category from database
    - Set it inside the product object before persist it

Exercises (3/3)

- Update the **addProduct.jsp** page
  - Add into the form a select field to choose the category
- Update the **showProduct.jsp** page
  - Display the category name of the product

Java Persistence API

GOOD PRACTICES

DAO & Factory patterns

Data Access Object Pattern

- Various methods are available to store information
  - Relational database
  - Object-oriented database
  - Flat files
  - LDAP
  - …

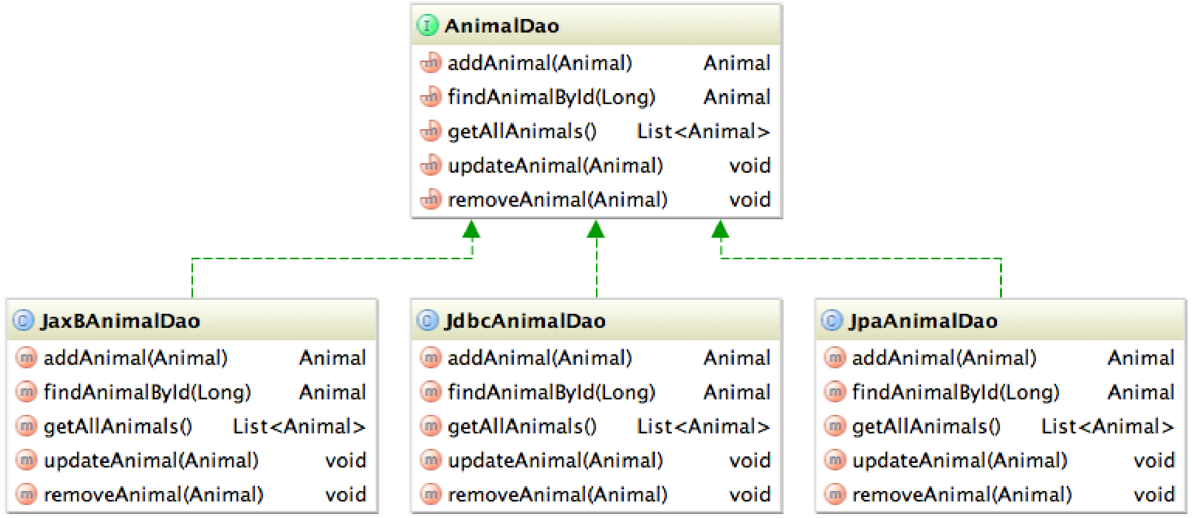
Data Access Object Pattern

- If your application change to another method
  - How to limit impact on the code ?
  - How to easily evolve the application ?

- Solution : add an abstract layer to centralize Data Access
  - With **Data Access Objects**

Data Access Object Pattern

- One interface define the necessary data access methods
- Several different implementations

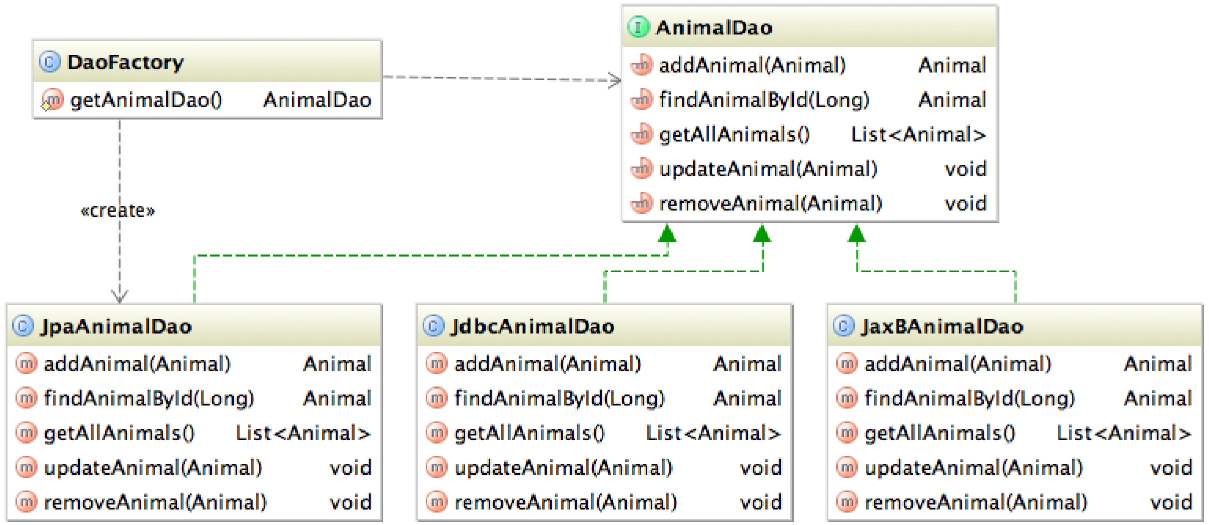

AnimalDao
addAnimal(Animal) Animal
findAnimalById(Long) Animal
getAllAnimals() List<Animal>
updateAnimal(Animal) void
removeAnimal(Animal) void
JaxBAnimalDao
JdbcAnimalDao
JpaAnimalDao

Data Access Object Pattern

- How to delete dependency between other classes and DAO implementations ?
  - Use **type inference**
    - Define your variables with the **interface type** instead of the implementation type
  - Use a **factory** to create DAO object
    - Delegate instance creation in a single point
    - When you'll want to change the implementation to use, just modify the factory !

Factory Pattern

- Delegate instance creation in a single point
  - When you'll want to change the implementation to use
    - Just modify your factory !



Good practices

Factory Pattern

- Example:

```
public class DaoFactory {
   //Private constructor prevent instantiation
   private DaoFactory(){}

   public static AnimalDao getAnimalDao() {
      return new JpaAnimalDao(
          PersistenceManager.getEntityManagerFactory());
   }
}
```

EntityManagerFactory

- Instances are expensive-to-create but thread safe…
- How to use only one instance ?
  - Create its own factory !
- How to destroy it when the web application ends ?
  - Create a **ServletContextListener** !

EntityManagerFactory

- Factory example 1/2:

```
public class PersistenceManager {
  private static EntityManagerFactory emf;

  // Lazy initialization
  public static EntityManagerFactory
      getEntityManagerFactory(){
    if(emf == null){
       emf = Persistence.createEntityManagerFactory("My-PU");
    }
    return emf;
  }
```

EntityManagerFactory

- Factory example 2/2:

```
   //Private constructor prevent instantiation
   private PersistenceManager(){}


   public static void closeEntityManagerFactory() {
     if(emf != null && emf.isOpen()) emf.close();
   }
}
```

ServletContextListener

- ServletContextListener example 1/2:

```
public class PersistenceAppListener
                          implements ServletContextListener {
    // Call on application initialization
    public void contextInitialized(ServletContextEvent evt){
        // Do nothing
    }

    // Call on application destruction
    public void contextDestroyed(ServletContextEvent evt) {
        PersistenceManager.closeEntityManagerFactory();
    }
}
```

ServletContextListener

- ServletContextListener example 2/2:

```
<web-app>
    ...
    <listener>
        <listener-class>
            com.supinfo.myapp.listener.PersistenceAppListener
        </listener-class>
    </listener>
    ...
</web-app>
```

Exercises (1/5)

- Create a new package
  - Name it **com.supinfo.supcommerce.util**
- Create one class inside
  - Name it **PersistenceManager**
  - Define a static method which return always the same instance of EntityManagerFactory
  - Define a static method to close this factory instance

Exercises (2/5)

- Create a new package
  - Name it **com.supinfo.supcommerce.listener**
- Create one class inside
  - Name it **PersistenceAppListener**
  - Implements **ServletContextListener**
  - In the **contextDestroyed(…)** method
    - Close your EntityManagerFactory instance
  - Declare your new listener in web.xml file or with the good Servlet 3.0 annotation

Exercises (3/5)

- Create a new package
  - Name it **com.supinfo.supcommerce.dao**
- Create two new interfaces inside
  - Name the first one **ProductDao**
    - Define all the data access methods you need to manage Product entities
  - Name the second one CategoryDao
    - Define all the data access **methods** you need to manage Category entities

Exercises (4/5)

- Create a new package
  - Name it **com.supinfo.supcommerce.dao.jpa**

  - Create two new classes inside
    - Name the first one **JpaProductDao**
      - Implements **ProductDao** interface
      - Define a constructor with an **EntityManagerFactory** parameter
  - Name the second one JpaCategoryDao
    - Implements CategoryDao interface
      - Define a constructor with an EntityManagerFactory parameter

Exercises (5/5)

- Create a class **DaoFactory**
  - Inside **com.supinfo.supcommerce.dao** package
  - Define a private constructor
  - Define two methods
    - One which return a new instance of **ProductDao**
    - Another one which return a new instance of **CategoryDao**

- Use your DAO instead of EntityManager in your Servlets!

Good practices

Summary

- We need to:
  - manage transaction manually
  - create DAO Factories
  - use a factory to create EntityManagers
  - close manually all our EntityManager instance
  - close manually our EntityManagerFactory
  - Preferably during application shutdown
- Ok, great…
  - Do you want an easier way to use JPA? Use EJB ;-)

The end

Sign of Success

Thanks for your attention