# JPA

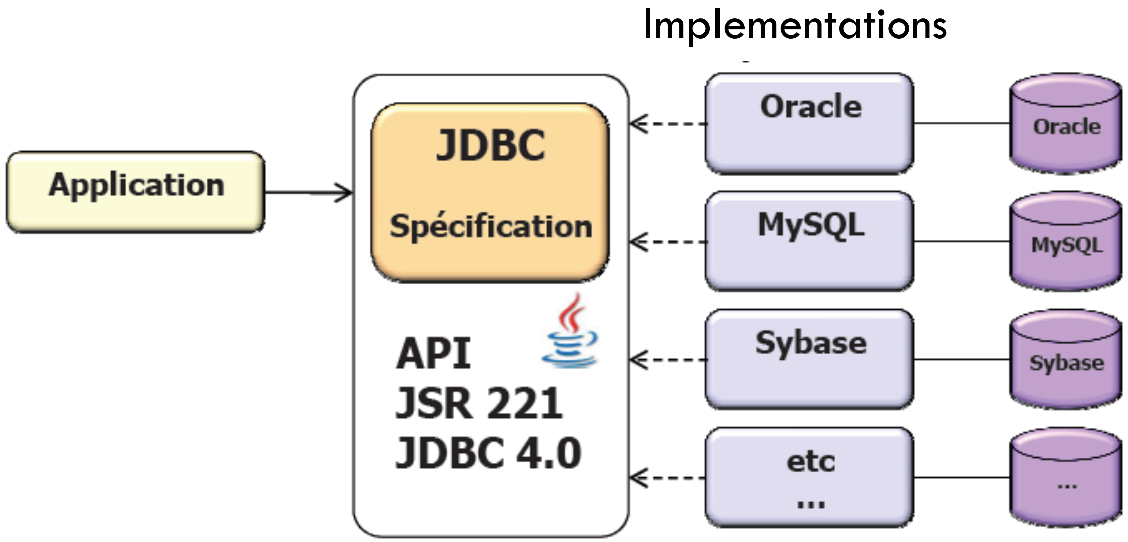
JAVA PERSISTENCE API

# INTRODUCTION

# FROM JDBC TO JPA

- For a long time the Java platform did not have more than one low-level API to manage access to relational databases:

**Java Database Connectivity (JDBC)**



# FROM JDBC TO JPA

Jdbc

- Relational database oriented
- Allow usage of SQL queries to consult and update “records” into “tables”
- “Low Level” API (Connections management, SQL requests, “ResultSet”, Transactions, etc)

Then, Higher level frameworks have appeared.

- Hibernate, TopLink, iBatis, …

A new specification have been added in the Java Platform: JDO (not very successful)

- JDO 1.0 – JSR 12 ( 2002 )
- JDO 2.0 – JSR 243 ( end of 2005 )

# FROM JDBC TO JPA

Under the influence of "ORM" (Object Relational Mapping) type frameworks and to respond to many criticisms regarding EJBs, a new specification is proposed and adopted:

## Java Persistence API (JPA)

- JPA 1.0
  - Java EE 5 ( JSR 220 / EJB 3.0 ) – May 2006
- JPA 2.0
  - Java EE 6 ( JSR 317 ) – Dec. 2009
- JPA 2.1
  - Java EE 7 (JSR 338) – Apr. 2013
- JPA 2.2
  - Under revision since 2017.

# API PERSISTENCE: DIFFERENT LEVELS

### ORM

- JPA (or Hibernate, TopLink, ...) ...
- High level, "Object Oriented"
- Trying to hide the complexity of the mapping
  - implement a lot of underlying concepts often misunderstood by the developers ("lazy / eager" loading, cache, "attached/detached" entities, "owning side "/" reverse side "links, ...)

### Others intermediate APIs

### JDBC API

- Low level API, “Records and SQL Oriented”. Addressing to the DB structure directly in SQL

# JDBC EXAMPLE

```
    Connection conn = null;
    try {
        conn = getConnection();
        PreparedStatement ps =
            conn.prepareStatement("SELECT .. FROM EMPLOYEE WHERE ...");
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            employee.setId(rs.getInt(1));
            employee.setName(rs.getString(2));
        }
        rs.close();
    } catch (SQLException e) {
        // ...
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (Exception ex) {
                // ...
            }
        }
    }
}
```

NATIVE SQL

MANUAL MAPPING

CONNECTION MANAGMENT

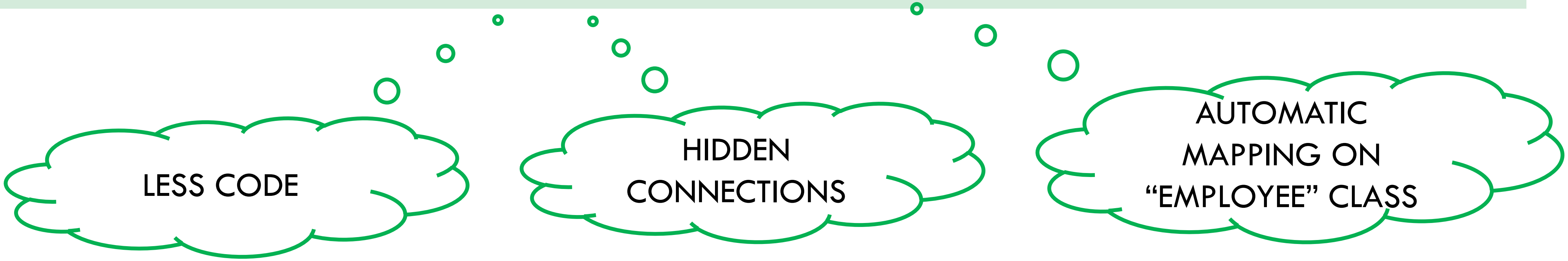
# ORM (JPA) EXAMPLE

## EMPLOYEES LIST

```
String displayAllQuery = "Select emp from Employee emp" ;
TypedQuery e = em.createQuery(displayAllQuery, Employee.class);
List <Employee> employees = e.getResultList();
for ( Employee emp : employees ) {
// ...
}
```

## SINGLE EMPLOYEE

```
Employee e = (Employee) em.find(Employee.class, id);
```



LESS CODE

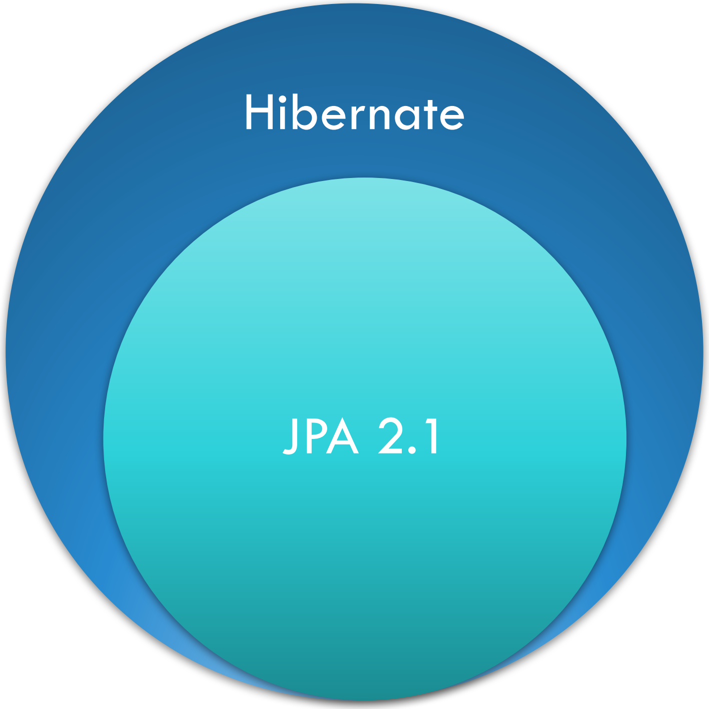
HIDDEN CONNECTIONS

AUTOMATIC MAPPING ON “EMPLOYEE” CLASS

# WHY JPA

Why using JPA instead of Hibernate, TopLink or else.

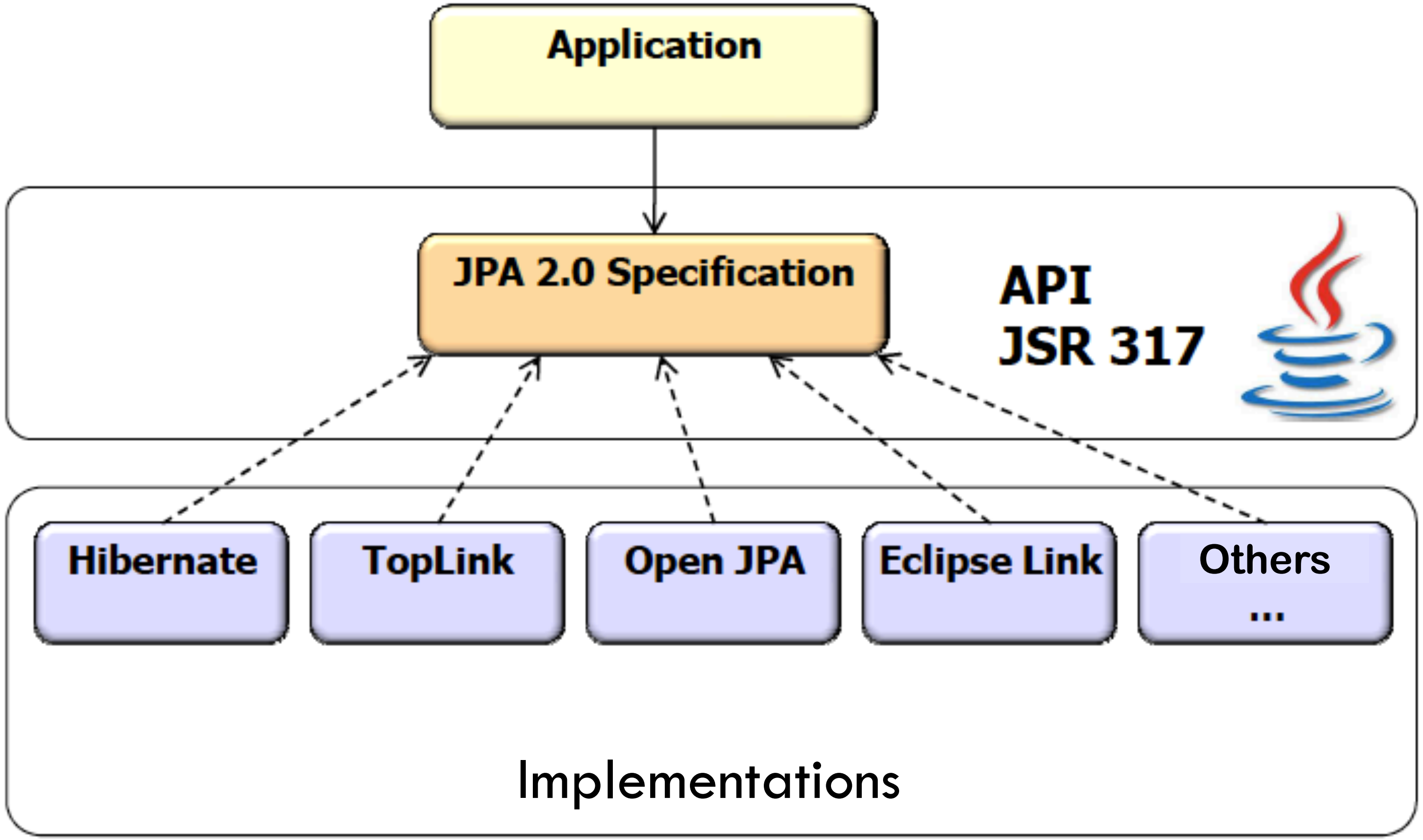
- JPA is part of JAVAEE Platform.
  - JPA is normalized
    - JPA 1.0
    - JPA 2.0
    - JPA 2.1
    - JPA 2.2 (Soon)
- However, some persistence frameworks can offer additional possibilities.



Hibernate

JPA 2.1

# SPECIFICATION, IMPLEMENTATION

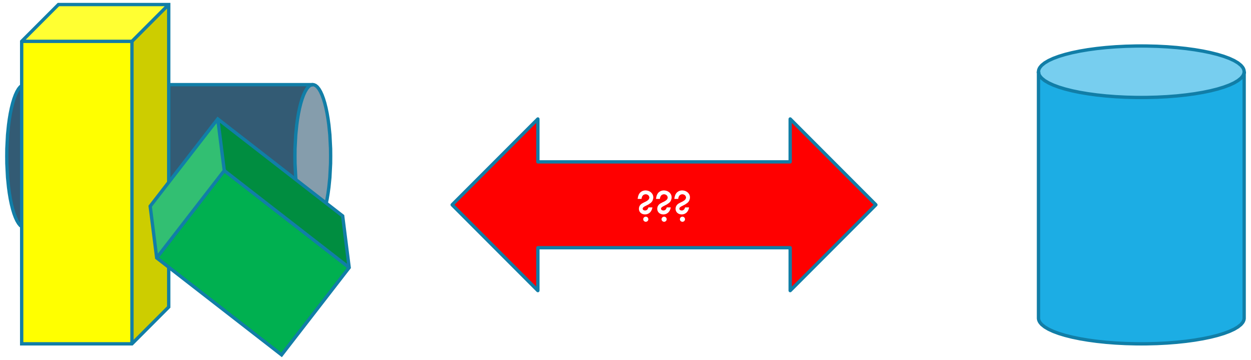


# MAPPING PROBLEMATICS “O/R”

Object Model ≠ Relational Model

Objects graph

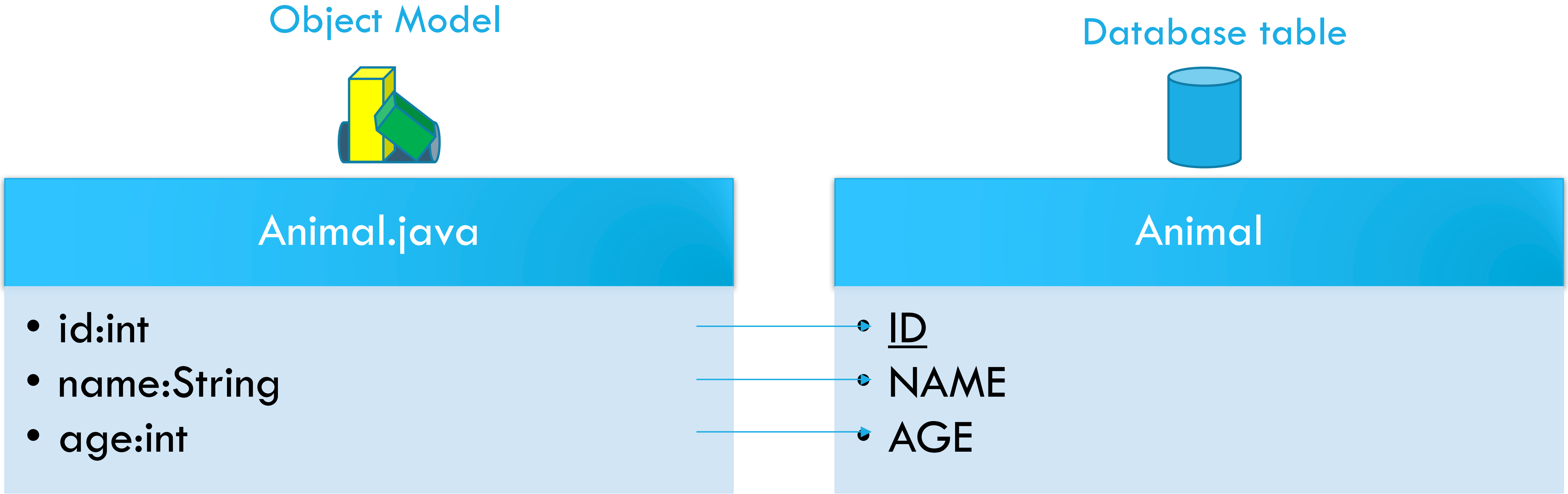
Relational Database



- Classes instances.
- References.
- No mandatory primary keys
- Inheritance

- Records in tables
- Relations (FK –> PK)

# SIMPLE MAPPING



Object Model

Animal.java

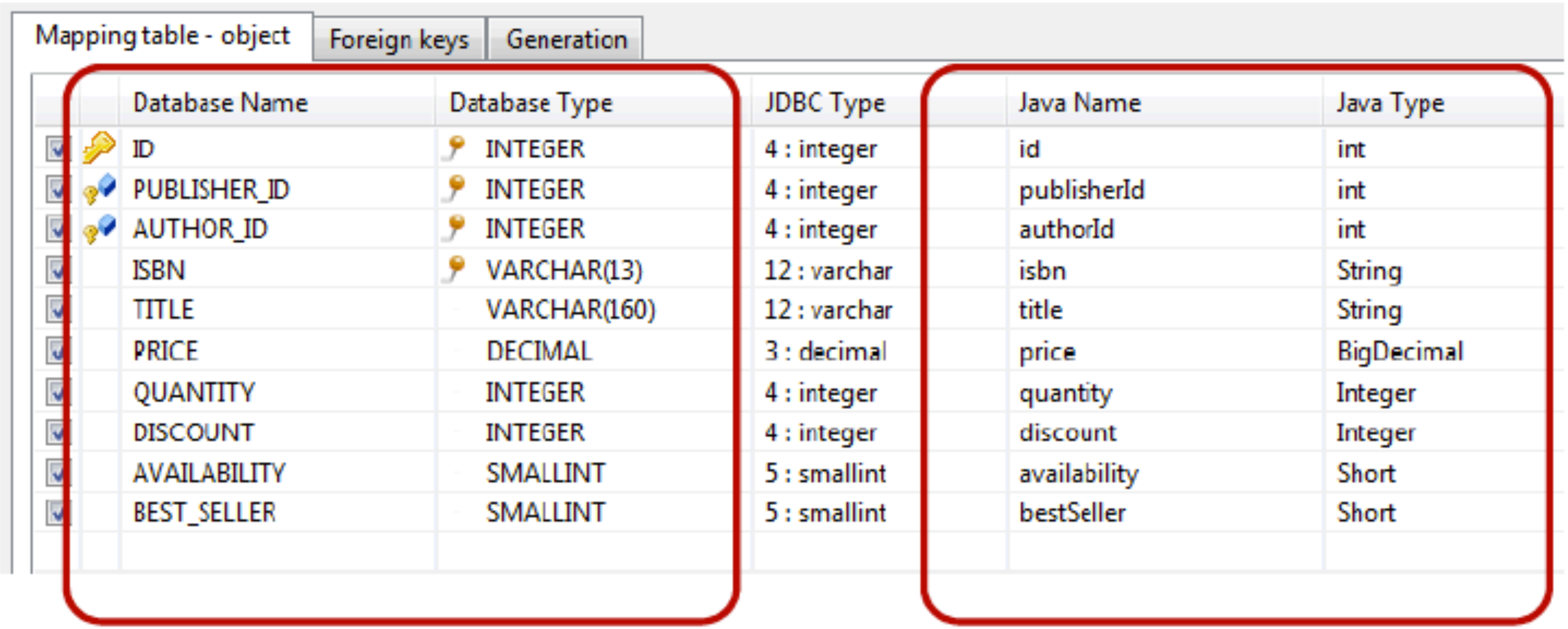
- id:int
- name:String
- age:int

Database table

Animal

- ID
- NAME
- AGE

# MAPPING EXAMPLE

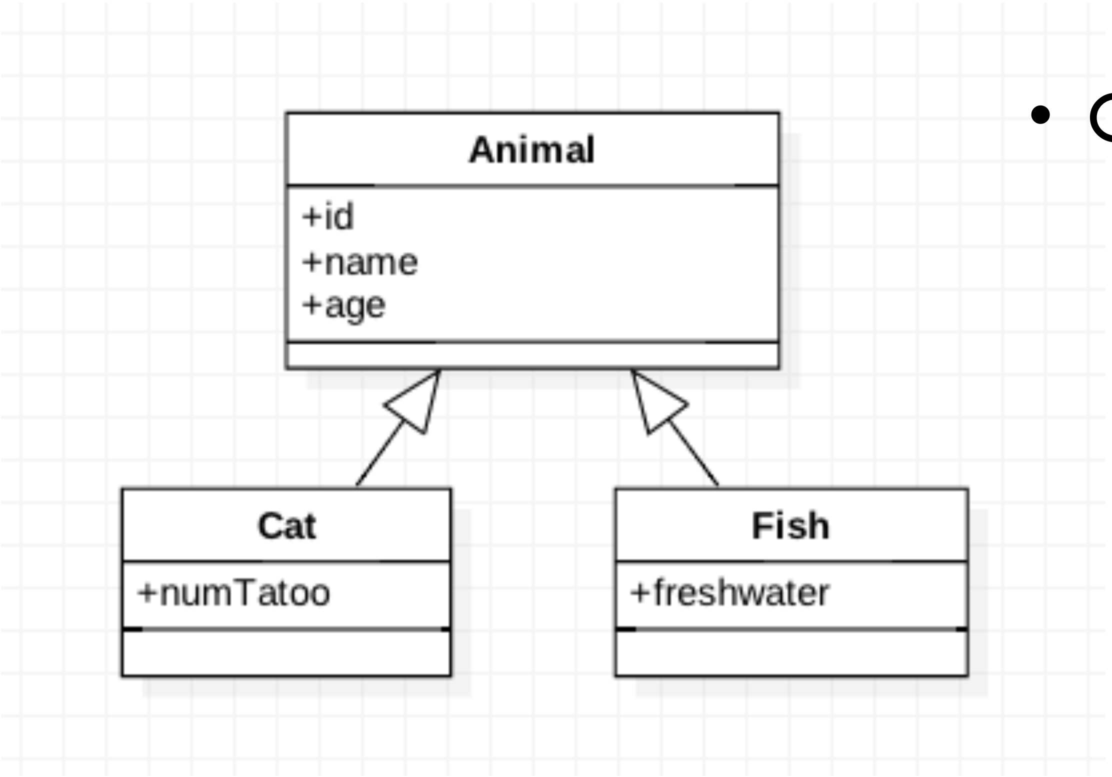


| | | Database Name | Database Type | JDBC Type | Java Name | Java Type |
|---|---|---|---|---|---|---|
| ☑ | 🔑 | ID | 🔑 INTEGER | 4 : integer | id | int |
| ☑ | 🔑 | PUBLISHER_ID | 🔑 INTEGER | 4 : integer | publisherId | int |
| ☑ | 🔑 | AUTHOR_ID | 🔑 INTEGER | 4 : integer | authorId | int |
| ☑ | | ISBN | 🔑 VARCHAR(13) | 12 : varchar | isbn | String |
| ☑ | | TITLE | VARCHAR(160) | 12 : varchar | title | String |
| ☑ | | PRICE | DECIMAL | 3 : decimal | price | BigDecimal |
| ☑ | | QUANTITY | INTEGER | 4 : integer | quantity | Integer |
| ☑ | | DISCOUNT | INTEGER | 4 : integer | discount | Integer |
| ☑ | | AVAILABILITY | SMALLINT | 5 : smallint | availability | Short |
| ☑ | | BEST_SELLER | SMALLINT | 5 : smallint | bestSeller | Short |

Table

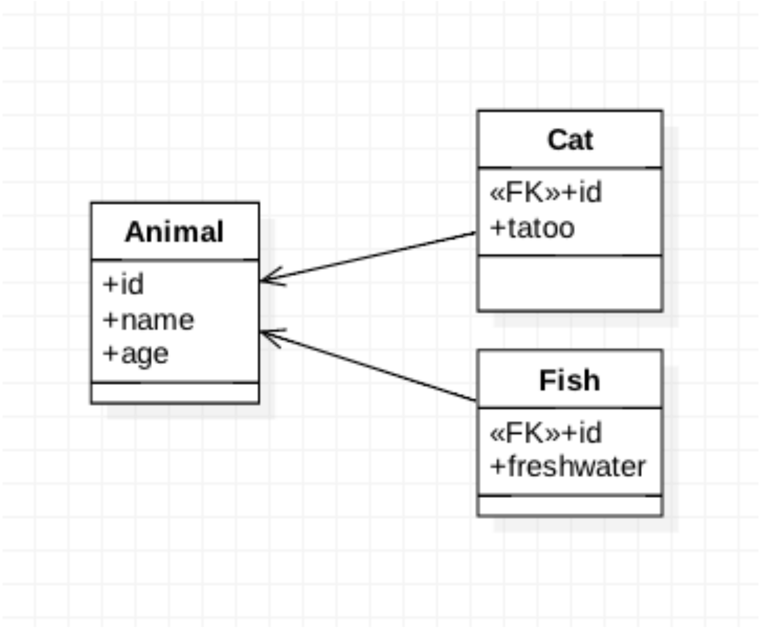
Object

# INHERITANCE CASE



- Classic inheritance case: 3 possibilities:
  - Vertical Inheritance : 3 tables
  - Horizontal Inheritance : 2 tables
  - Filtering by Type: 1 Table

# INHERITANCE CASE



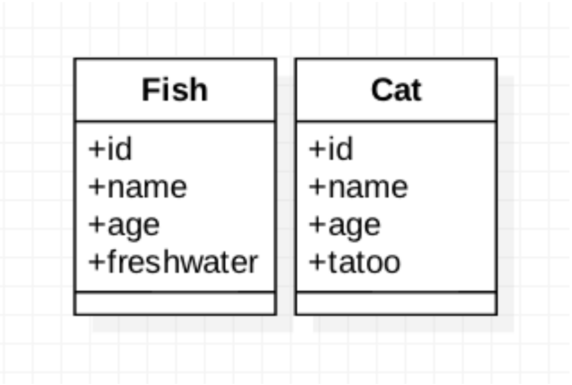
**Animal**
+id
+name
+age

**Cat**
«FK»+id
+tatoo

**Fish**
«FK»+id
+freshwater

## VERTICAL INHERITANCE

- JOIN COST

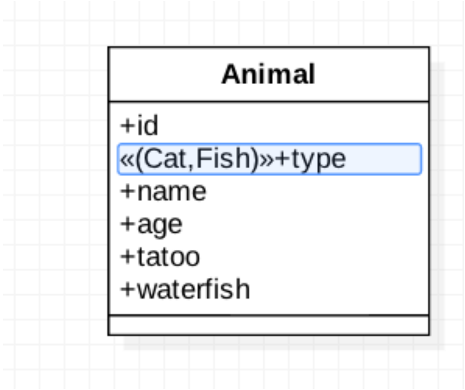


**Fish**
+id
+name
+age
+freshwater

**Cat**
+id
+name
+age
+tatoo

## HORIZONTAL INHERITANCE

- PROBLEM WITH UNICITY OF THE ID
- DUPLICATION OF ATTRIBUTES

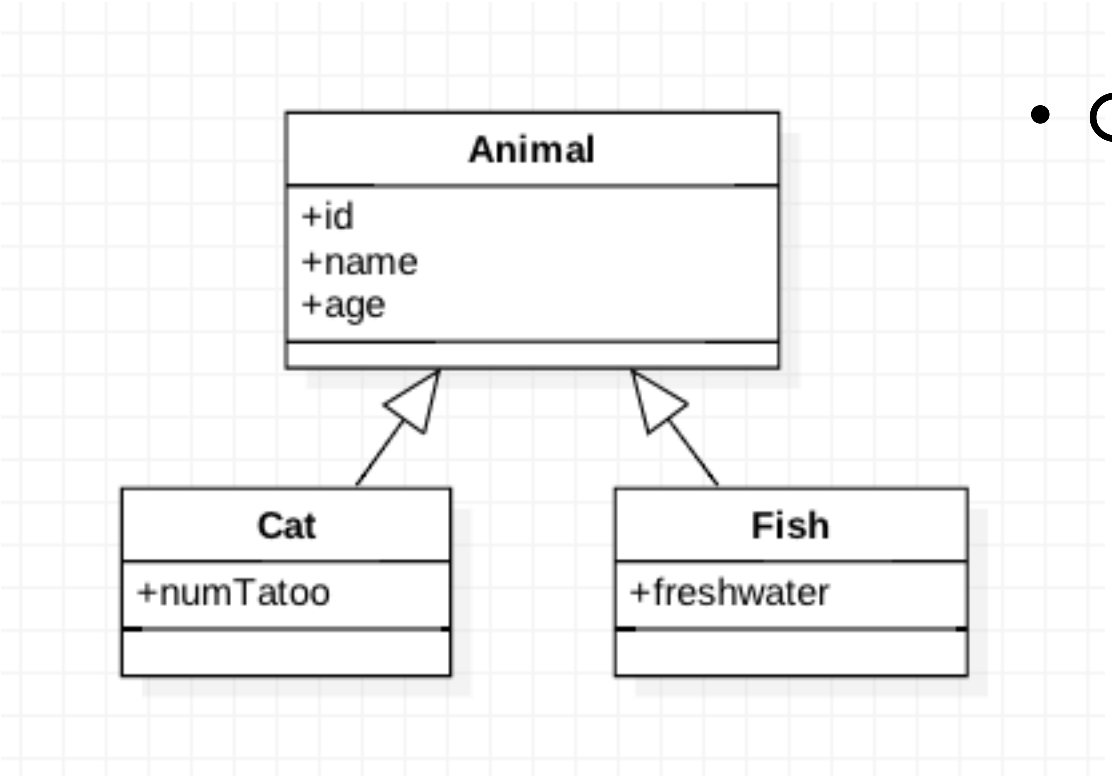


**Animal**
+id
«(Cat,Fish)»+type
+name
+age
+tatoo
+waterfish

## FILTERING BY TYPE

- TYPE MIXING

# INHERITANCE CASE

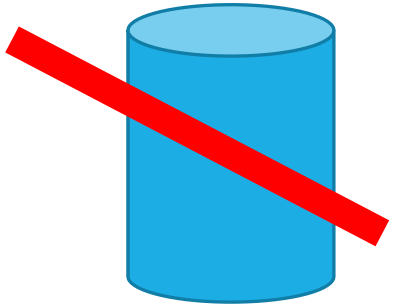


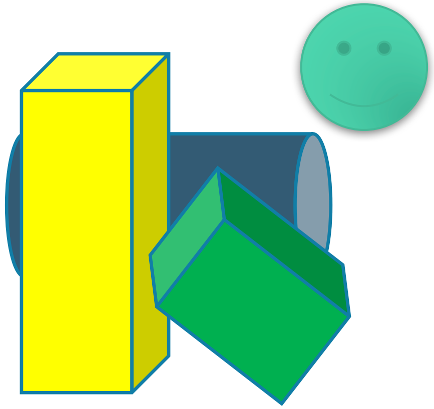
- Classic inheritance case: 3 possibilities:
  - Vertical Inheritance : 3 tables
  - Horizontal Inheritance : 2 tables
  - Filtering by Type: 1 Table

# THE GREAT PRINCIPLES OF JPA

# OBJECTIVE: STAYING AT AN OBJECT LEVEL

- JPA gives the impression to the developer to work with an object oriented database ("Object database").
- JPA masks the whole Relational "plumbing”.
- Connections to the database are not visible (JDBC "Connection" objects)
- In the usual cases the developer never use the native SQL (the use of SQL queries however, remains possible for the specific cases)





# MAPPING BY ANNOTATIONS

```
@Entity
@Table(name="user")
public class User {

     @Id
     @GeneratedValue(strategy=GenerationType.IDENTITY)
     private int id;

     @Column
     private String nom;

     @Column
     private String password;
```

We can also map our classes using an xml mapping file, however, this method is not that much used nowadays.

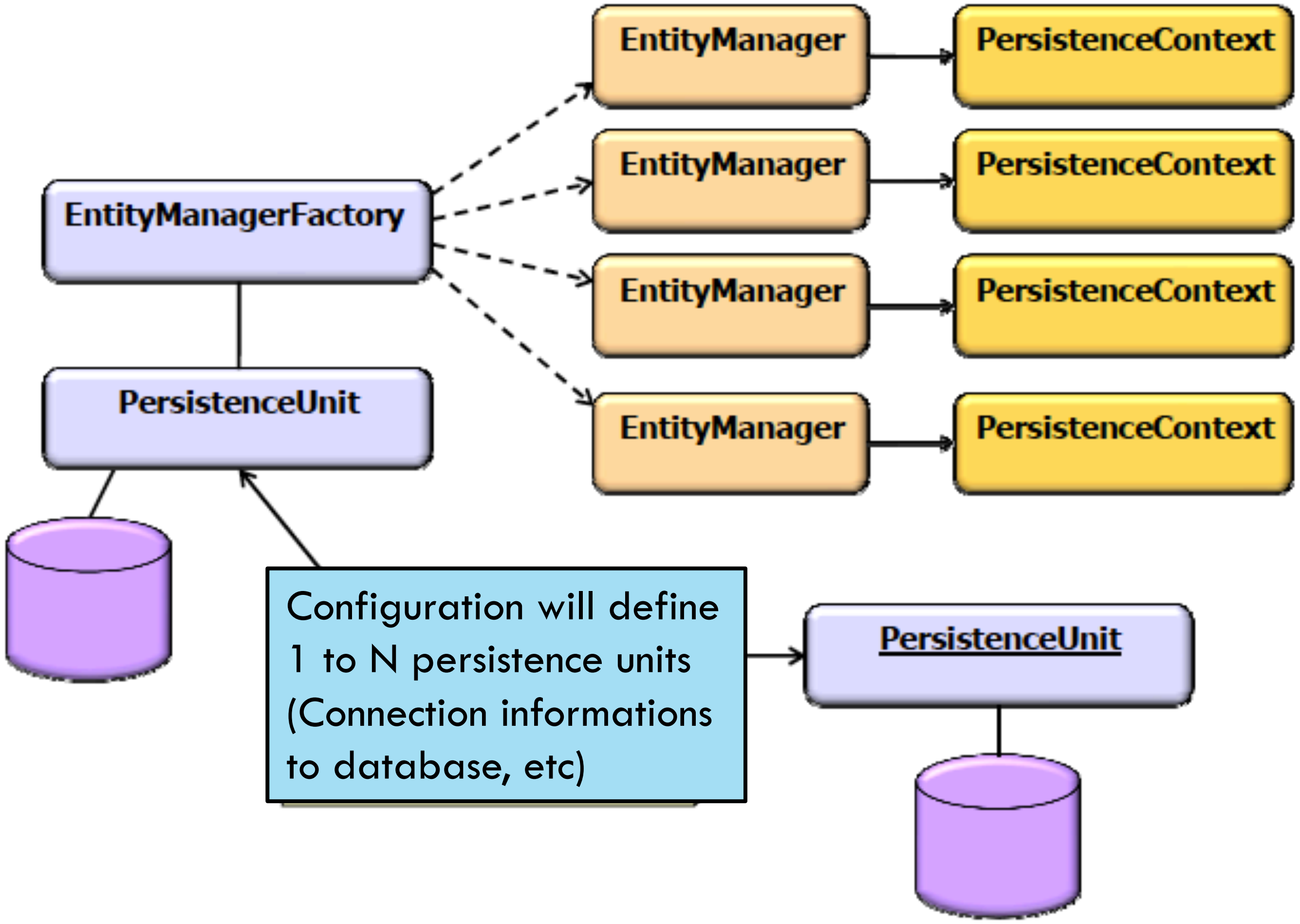
# A FEW OBJECTS TO MANAGE EVERYTHING

Each database will be described as a “Persistence Unit” (both strategies can be used to describe the database : Annotation or Configuration (XML).

Each ”Persistence Unit” has a corresponding “EntityManagerFactory”

Persistence operations rely on a central object: "EntityManager” which is provided by the "EntityManagerFactory" of the database to be addressed.

# MAIN OBJECTS



Configuration will define 1 to N persistence units (Connection informations to database, etc)

# PERSISTENCE.XML

Located in the META-INF folder

```
      <persistence-unit name="ecommerce" transaction-type="RESOURCE_LOCAL">
           <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
           <properties>
                <property name="hibernate.connection.driver_class" value="org.postgresql.Driver" />
                <property name="hibernate.connection.url" value="jdbc:postgresql://localhost:5432/ecommerce" />
                <property name="hibernate.connection.username" value="postgres" />
                <property name="hibernate.connection.password" value="admin" />
                <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
                <property name="hibernate.show_sql" value="true" />
                <property name="hibernate.hbm2ddl.auto" value="update" />
           </properties>
      </persistence-unit>
```

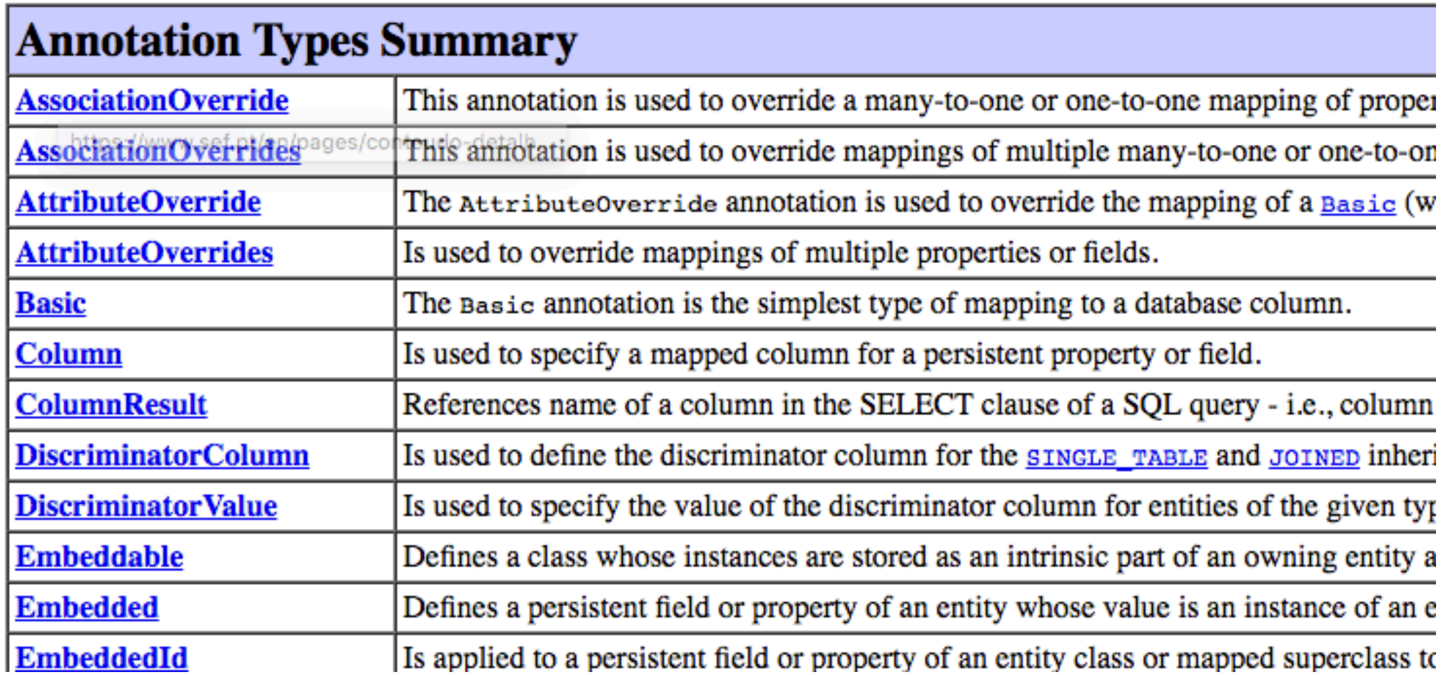
# STARTING AND ENDING A PROCESS

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("ecommerce");

// Getting an instance of EntityManager
EntityManager em = emf.createEntityManager();
// Usage of the "EntityManager" to fetch elements for example
// .....
// Closing the "EntityManager"
em.close();
//Closing the "EntityManagerFactory"
emf.close();
```

# O/R MAPPING WITH JPA

# JPA – MAPPING O/R

JPA mapping is based on annotations from the »javax.persistence»  package



| Annotation Types Summary | |
|---|---|
| AssociationOverride | This annotation is used to override a many-to-one or one-to-one mapping of proper |
| AssociationOverrides | This annotation is used to override mappings of multiple many-to-one or one-to-on |
| AttributeOverride | The `AttributeOverride` annotation is used to override the mapping of a `Basic` (w |
| AttributeOverrides | Is used to override mappings of multiple properties or fields. |
| Basic | The `Basic` annotation is the simplest type of mapping to a database column. |
| Column | Is used to specify a mapped column for a persistent property or field. |
| ColumnResult | References name of a column in the SELECT clause of a SQL query - i.e., column |
| DiscriminatorColumn | Is used to define the discriminator column for the `SINGLE_TABLE` and `JOINED` inheri |
| DiscriminatorValue | Is used to specify the value of the discriminator column for entities of the given typ |
| Embeddable | Defines a class whose instances are stored as an intrinsic part of an owning entity a |
| Embedded | Defines a persistent field or property of an entity whose value is an instance of an e |
| EmbeddedId | Is applied to a persistent field or property of an entity class or mapped superclass to |

# JPA – MAPPING O/R

- Mapping of the Entity (java class)
    - Association of the class to the table

```
@Entity
@Table(name="employees")
public class Employee {
```

- Mapping of fields (java attributes)

```
@Entity
public class Employee {

     @Id
     private Long id;

     @Column(name="empName")
     private String name;
```

Annotation can be placed on a field level or on accessors (setXxx())

# JPA – MAPPING O/R

Primary Key : @Id

```
@Id
@Column(name="EMP_ID")
private int id;
```

BLOB/CLOB : @Lob

```
@Basic(fetch=FetchType.LAZY)
@Lob
@Column(name="PIC")
private byte[] picture;
```

Date Type

```
@Temporal(TemporalType.DATE) // .DATE .TIME or .TIMESTAMP
@Column(name="START_DATE")
private Date startDate;
```

# JPA – MAPPING O/R

AUTO: Generation strategy defined by JPA

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private int id;
```

IDENTITY: ID generated by an auto-increment strategy.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

TABLE: ID managed in a table.

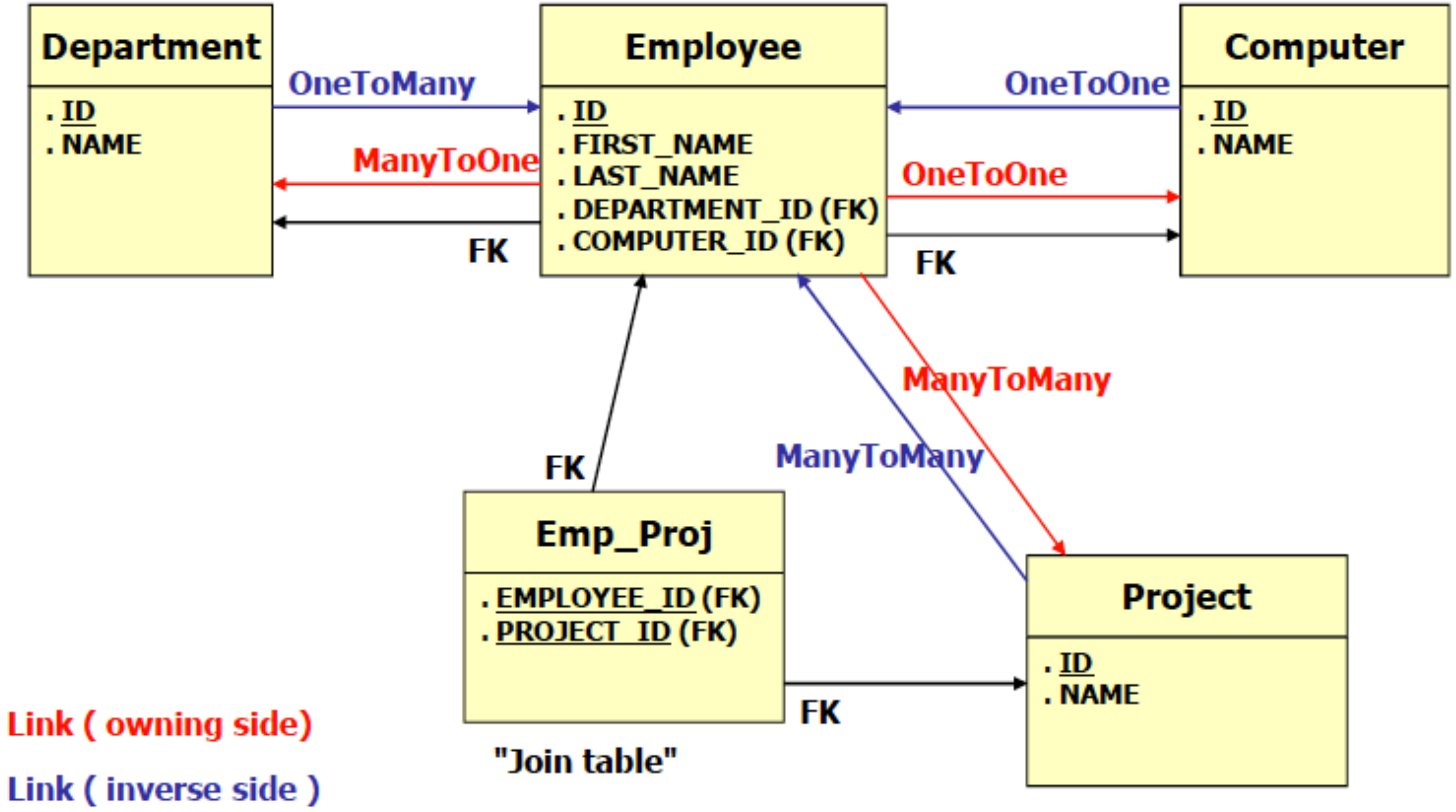
```
@Id
@TableGenerator(name="EmpGen", table="x",
     pkColumnName="x" , valueColumnName="x" )
@GeneratedValue( generator = "EmpGen" )
private int id;
```

SEQUENCE: ID managed by a Sequence.

```
@Id
@SequenceGenerator(name="EmpGen", sequenceName="SEQ1")
@GeneratedValue( generator = "EmpGen" )
private int id;
```

# JPA – MAPPING O/R : LINKS

Link types in JPA



# JPA – MAPPING O/R : LINKS

A relationship between 2 entities is based on **2 links** (a link for each direction)

- Each link have a **direction** and a **cardinality.**

**One To One**

**One To Many**

**Many To One**

**Many To Many**

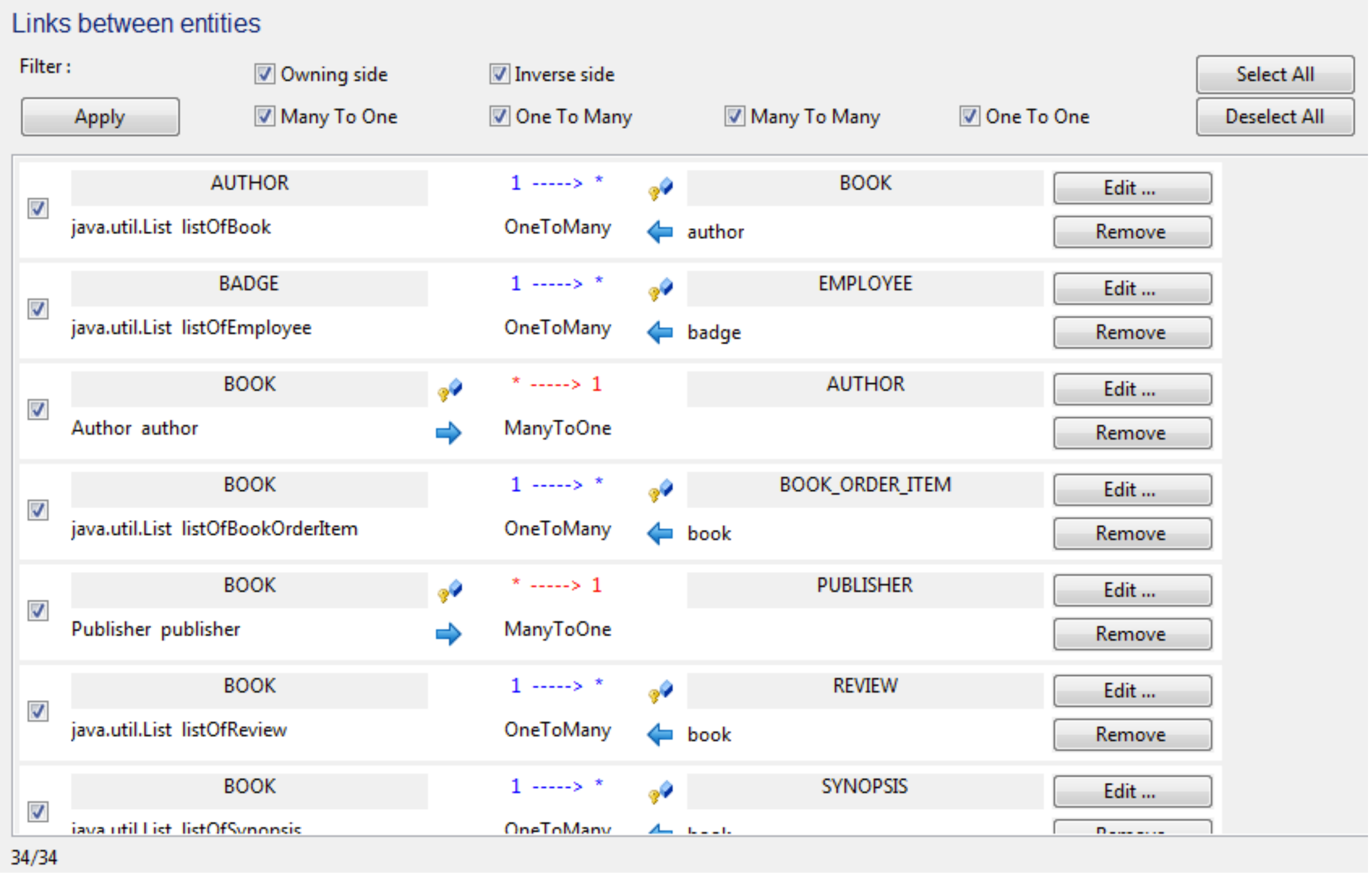
## Owning Side

- The one who own the foreign key

## Inverse Side

- The side ”referenced” by the “owner”
- *“The owner has the power”*

# EXAMPLE



Links between entities

Filter :

☑ Owning side ☑ Inverse side

☑ Many To One ☑ One To Many ☑ Many To Many ☑ One To One

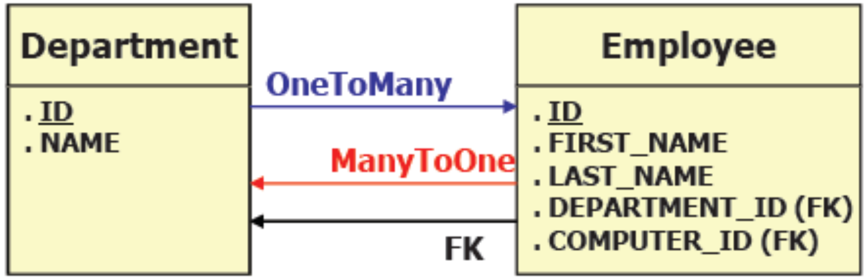
| | | | |
|---|---|---|---|
| ☑ | AUTHOR<br>java.util.List listOfBook | 1 -----> *<br>OneToMany | BOOK<br>author |
| ☑ | BADGE<br>java.util.List listOfEmployee | 1 -----> *<br>OneToMany | EMPLOYEE<br>badge |
| ☑ | BOOK<br>Author author | * -----> 1<br>ManyToOne | AUTHOR |
| ☑ | BOOK<br>java.util.List listOfBookOrderItem | 1 -----> *<br>OneToMany | BOOK_ORDER_ITEM<br>book |
| ☑ | BOOK<br>Publisher publisher | * -----> 1<br>ManyToOne | PUBLISHER |
| ☑ | BOOK<br>java.util.List listOfReview | 1 -----> *<br>OneToMany | REVIEW<br>book |
| ☑ | BOOK<br>java.util.List listOfSynopsis | 1 -----> *<br>OneToMany | SYNOPSIS<br>book |

34/34

# JPA - MAPPING O/R - LINKS

Exemple : **Many To One ( Owning Side )**

```
@Entity
public class Employee {

     @Id
     private int id;

     @ManyToOne
     @JoinColumn(name="DEPARTMENT_ID")
     private Department department;
     // …
}
```

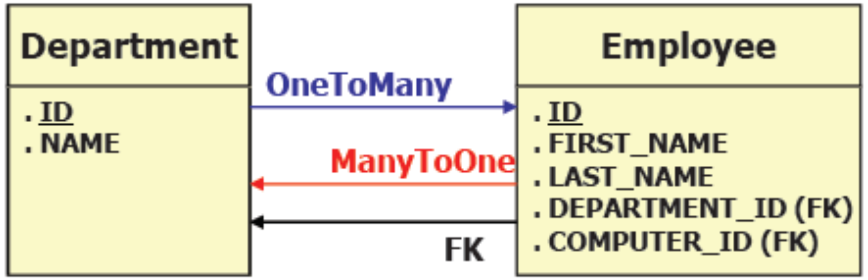


# JPA - MAPPING O/R - LINKS

Exemple : **One To Many ( Inverse Side )**

```
@Entity
public class Department {
     @Id
     private int id;
     private String name;

     @OneToMany(mappedBy="department")
     private Collection<Employee> employees;
// …
}
```

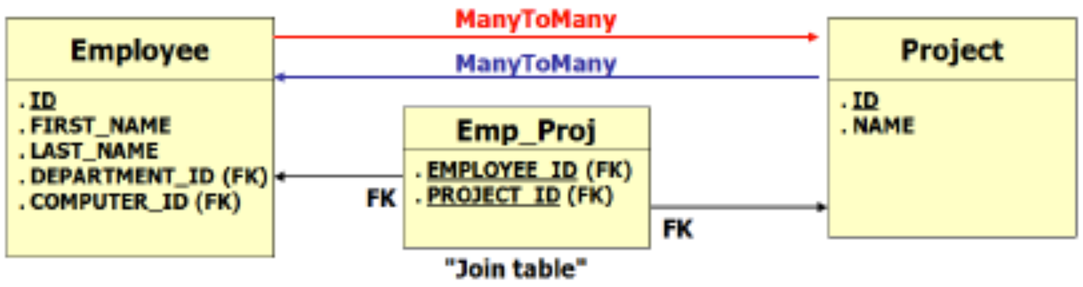
Attribute in the owning side !!



# JPA - MAPPING O/R - LINKS

Exemple : **Many To Many**

```
@Entity
public class Employee {
      @Id
      private int id;
      //...

      @ManyToMany
      @JoinTable( name="EMP_PROJ",
      joinColumns = @JoinColumn(name="EMPLOYEE_ID"),
      inverseJoinColumns = @JoinColumn(name="PROJECT_ID"))
      private Collection<Project> projects;
      // ...
}
```



# JPA - MAPPING O/R - LOADING

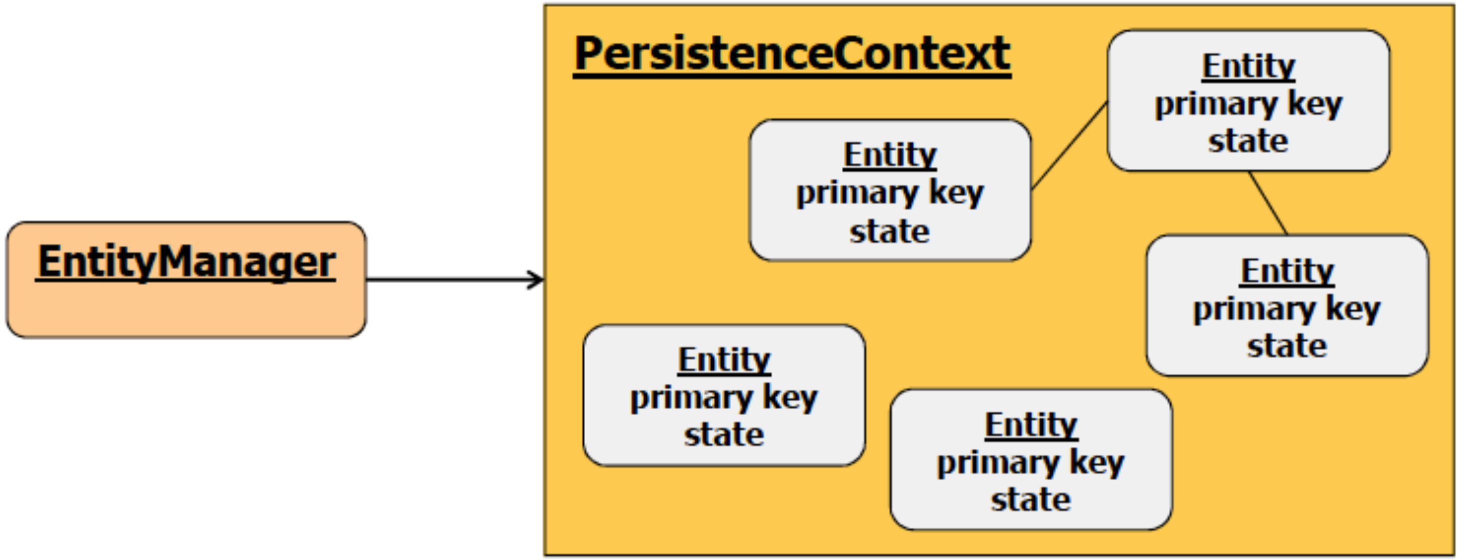
- When an entity is loaded into the"PersistenceContext", its links can be:
  - Immediately loaded: « **Eager Loading** ».
  - Loaded later, only whent he application will use them: « **Lazy Loading** «

```
@Entity
public class Employee {
     @Id
     private int id;
     @OneToOne(fetch = FetchType.LAZY)
     private ParkingSpace parkingSpace;
// …
}
```

# JPA | ARCHITECTURE

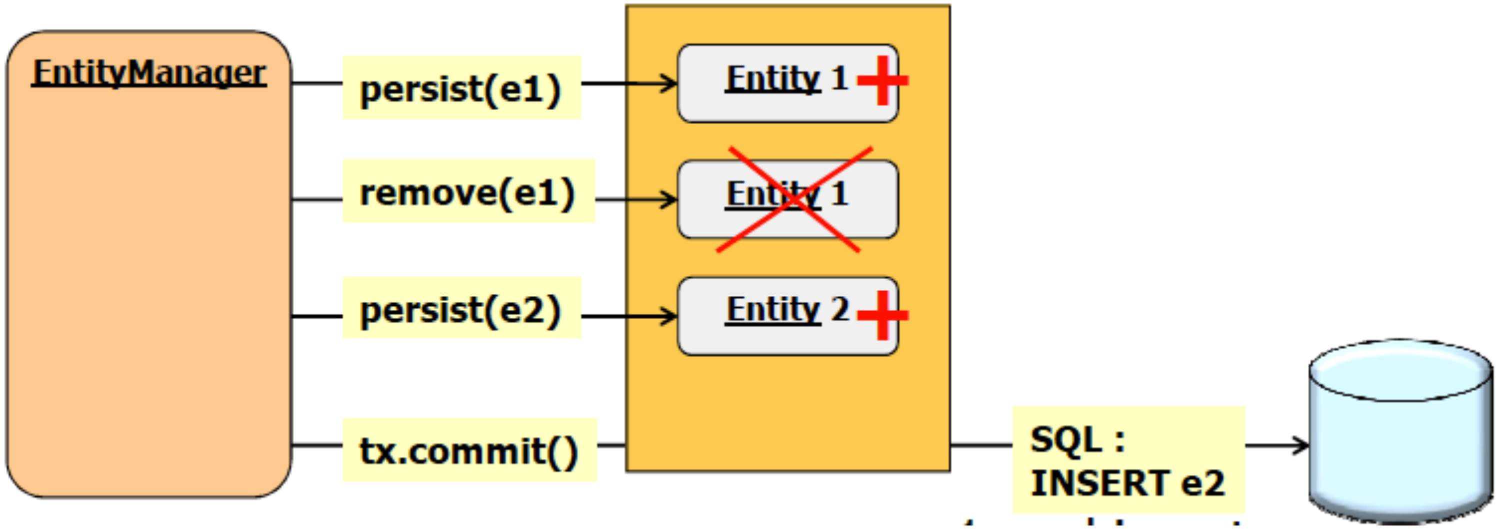
# THE PERSISTENCE CONTEXT

- The "PersistenceContext" is a storage space in memory that contains « entities »
- Each entity has a state and is identified by its primary key (it’s impossible to have 2 instances of a same class with the same primary key)

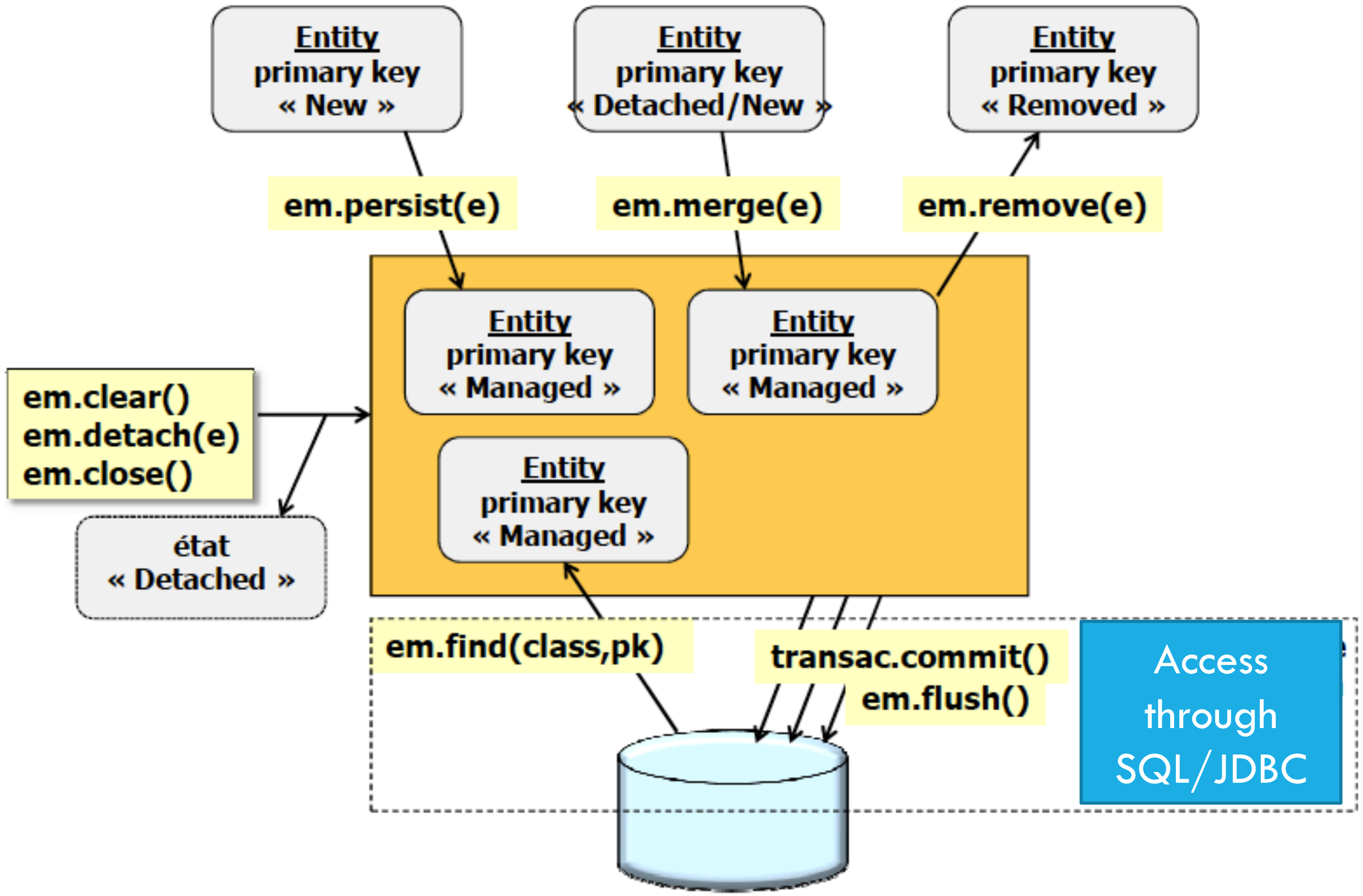


# THE PERSISTENCE CONTEXT

- The EntityManager manages the state of instances whose he has charge of ("Managed" objects in the"PersistenceContext")
- He decides when and how to updates the database



# ENTITIES MANAGMENT

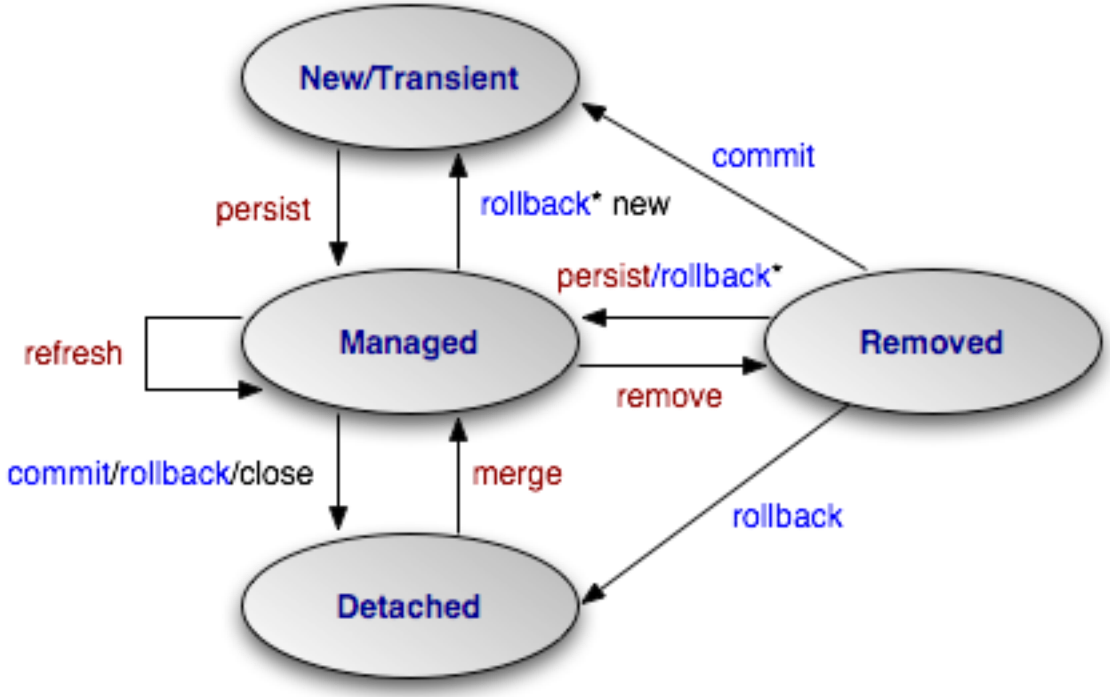


# DIFFERENT STATES OF AN ENTITY

Each entity have a state that can be:

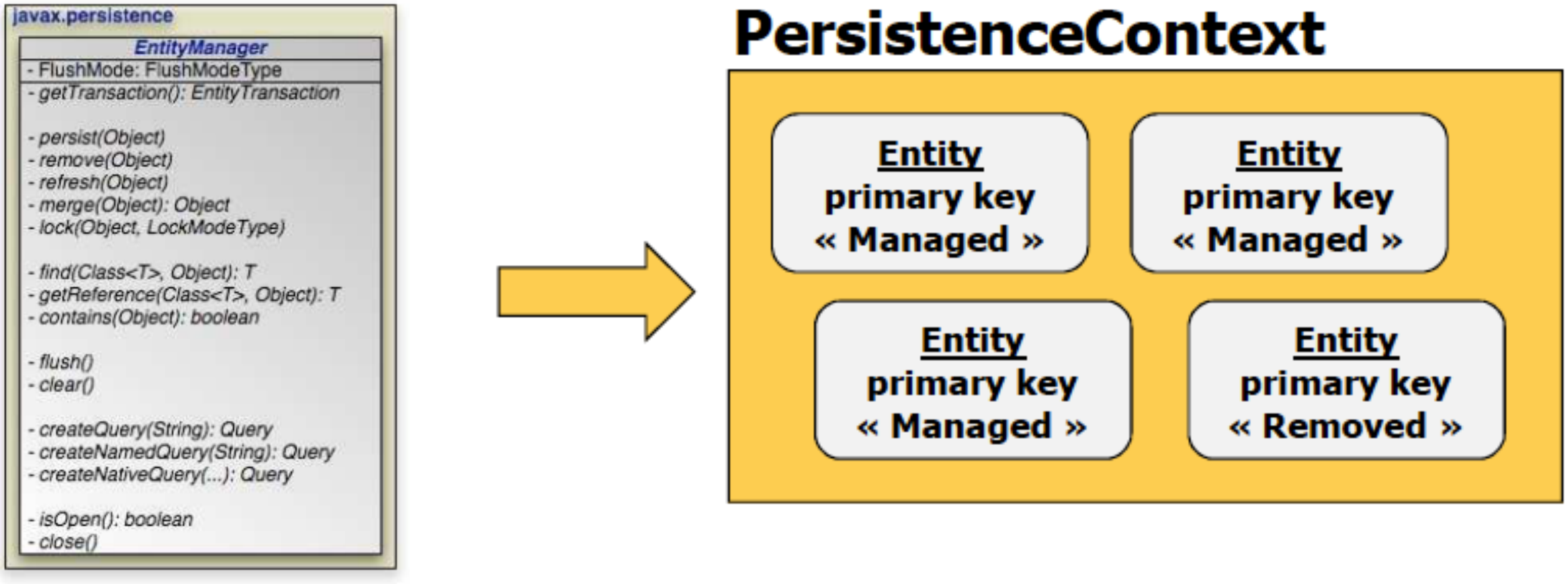
- **NEW** (or TRANSIANT) : Not managed.
- **MANAGED**: Managed  .
- **REMOVED**: Deleted
- **DETACHED**: Detached, not managed anymore

This state is changing according calls on different methods of the EntityManager



# ENTITYMANAGER

This is the **EntityManager** who manages all the persistence operations on entities



javax.persistence

*EntityManager*

- FlushMode: FlushModeType
- *getTransaction(): EntityTransaction*

- *persist(Object)*
- *remove(Object)*
- *refresh(Object)*
- *merge(Object): Object*
- *lock(Object, LockModeType)*

- *find(Class<T>, Object): T*
- *getReference(Class<T>, Object): T*
- *contains(Object): boolean*

- *flush()*
- *clear()*

- *createQuery(String): Query*
- *createNamedQuery(String): Query*
- *createNativeQuery(...): Query*

- *isOpen(): boolean*
- *close()*

**PersistenceContext**

**Entity**
**primary key**
**« Managed »**

**Entity**
**primary key**
**« Managed »**

**Entity**
**primary key**
**« Managed »**

**Entity**
**primary key**
**« Removed »**

# ENTITYMANAGER: MAIN METHODS

**persist(entity):** Adding a new entity

**merge(entity):** Updating a new Entity (Adding if not existing)

**remove(entity):** Deleting an Entity

**find(type, key):** Search for an Entity with its ID

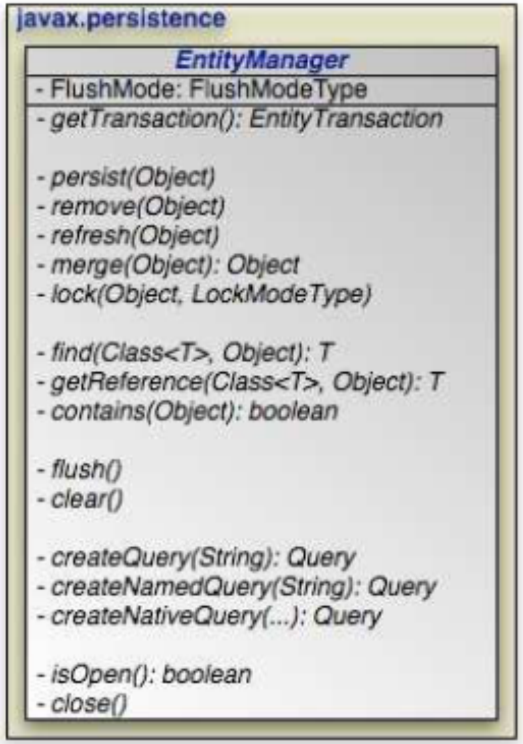
**refresh(entity):** Refresh the Entity from the DB

**flush():** Force updates into DB

**clear():** Emptying the persistent context

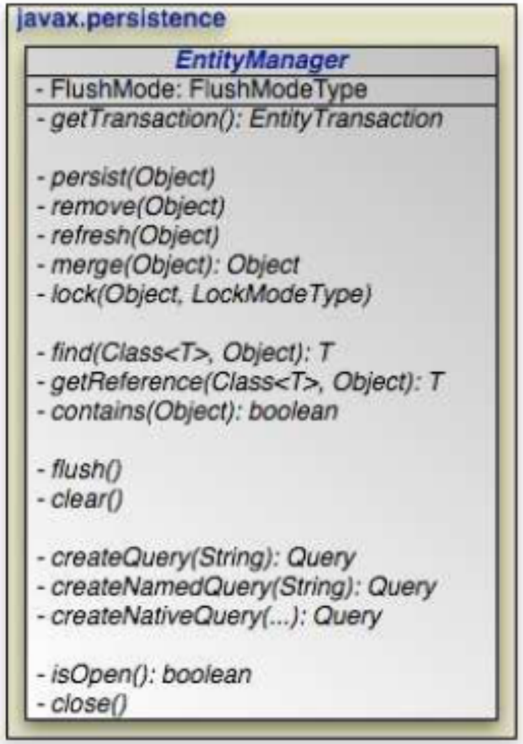
**getTransaction(): Get the current transaction**

**close():** Close the EntityManager (Do not commit)


javax.persistence
EntityManager
- FlushMode: FlushModeType
- getTransaction(): EntityTransaction
- persist(Object)
- remove(Object)
- refresh(Object)
- merge(Object): Object
- lock(Object, LockModeType)
- find(Class<T>, Object): T
- getReference(Class<T>, Object): T
- contains(Object): boolean
- flush()
- clear()
- createQuery(String): Query
- createNamedQuery(String): Query
- createNativeQuery(...): Query
- isOpen(): boolean
- close()

# ENTITYMANAGER

An "entity" type parameter is expected by most of the methods (persist, merge, remove, ...)
This parameter must be an instance of a class annotated with "@Entity" (with the mapping of java fields to columns of the table)



javax.persistence

*EntityManager*

- FlushMode: FlushModeType

- *getTransaction(): EntityTransaction*

- *persist(Object)*
- *remove(Object)*
- *refresh(Object)*
- *merge(Object): Object*
- *lock(Object, LockModeType)*

- *find(Class<T>, Object): T*
- *getReference(Class<T>, Object): T*
- *contains(Object): boolean*

- *flush()*
- *clear()*

- *createQuery(String): Query*
- *createNamedQuery(String): Query*
- *createNativeQuery(...): Query*

- *isOpen(): boolean*
- *close()*

# ENTITY MANAGER: BASE OPERATIONS

# IDEA OF "CRUD"

CRUD:

- C : CREATE
- R : READ
- U : UPDATE
- D : DELETE

CRUD with SQL:

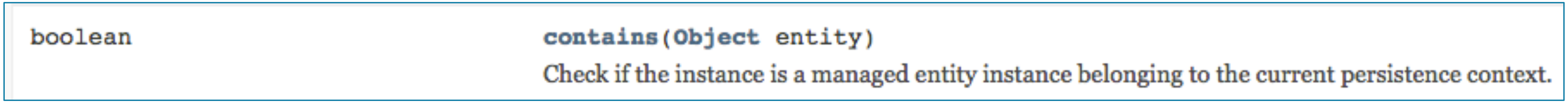
- C : Insert into … values
- R : Select … from … where
- U : Update … set … where
- D : Delete from … where

With JPA, it's a bit different!

# IDEA OF "CRUD"

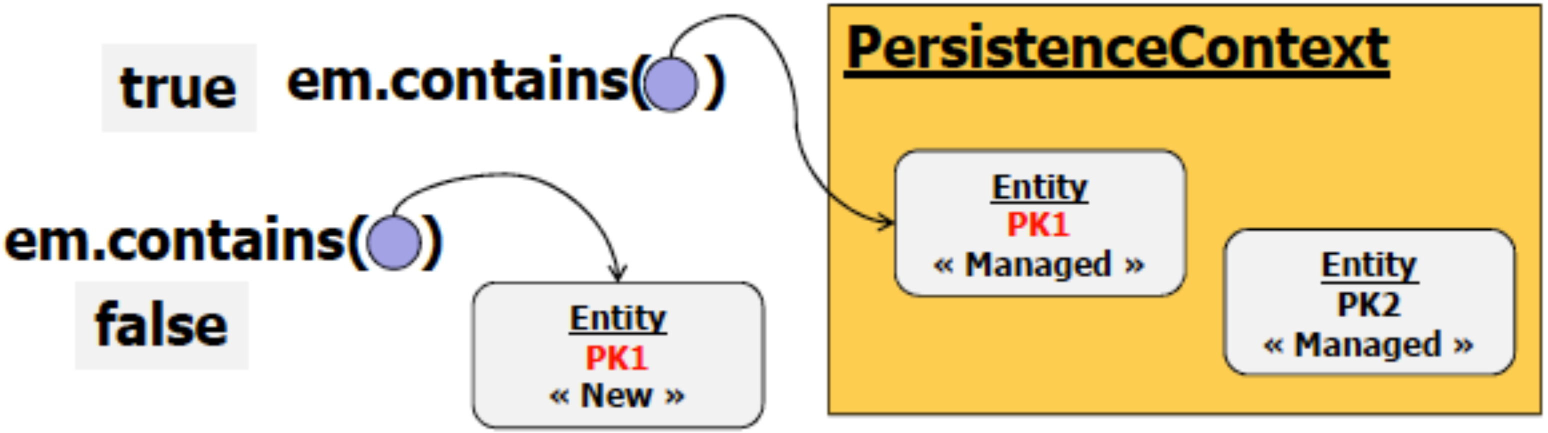
- Update methods such as persist(), merge() or remove() are not realizing immediate action into database
- Those updates are realized into the "PersistenceContext" (In Memory)
- The EntityManager then decide how and when it will affect the database according the value of ”FlushModeType" parameter (AUTO or COMMIT)
- There is no direct correspondence between JPA and an SQL order.

# ENTITYMANAGER: CONTAINS(E)

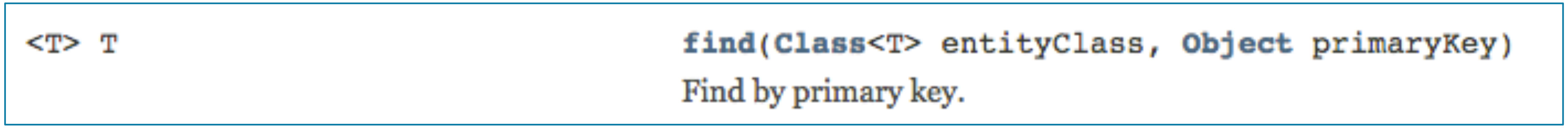


| | |
|---|---|
| boolean | **contains(Object entity)**<br>Check if the instance is a managed entity instance belonging to the current persistence context. |

- The test is made on the instance and not on the primary key!

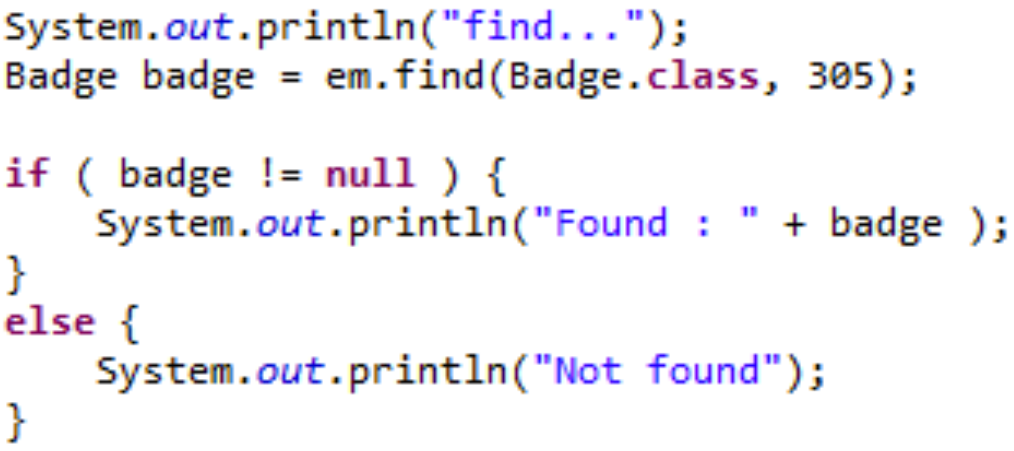


# ENTITYMANAGER: FIND(E,ID)



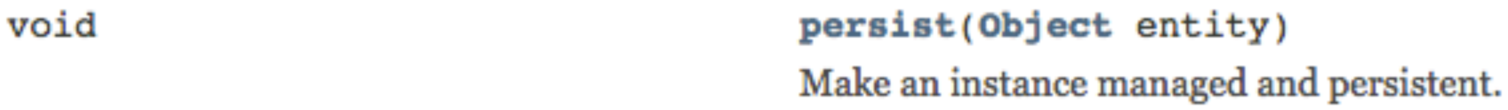
| | |
|---|---|
| `<T> T` | `find(Class<T> entityClass, Object primaryKey)`<br>Find by primary key. |

- Search for an Entity according to its Primary Key and load it in the Persistence Context
- Loaded entity state is “Managed”



```java
System.out.println("find...");
Badge badge = em.find(Badge.class, 305);

if ( badge != null ) {
    System.out.println("Found : " + badge );
}
else {
    System.out.println("Not found");
}
```

# ENTITYMANAGER: PERSIST(E)



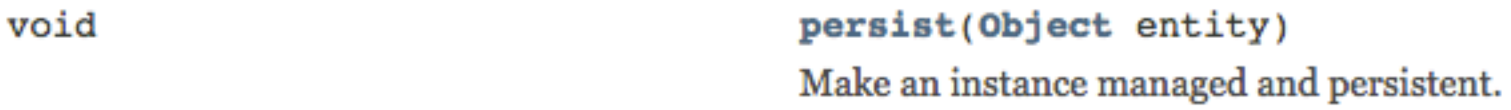
void persist(Object entity)
Make an instance managed and persistent.

- Have to be used in an active transaction (otherwhise: TransactionRequiredException )
- Comportment:
  - Change of state: “Managed”
    - If the state is “New” : Turns to “Managed”
    - If state is “Managed” : Ignored
    - If state is “Removed” : Turns to “Managed”
    - If state is “Detached” : **IllegalArgumentException**

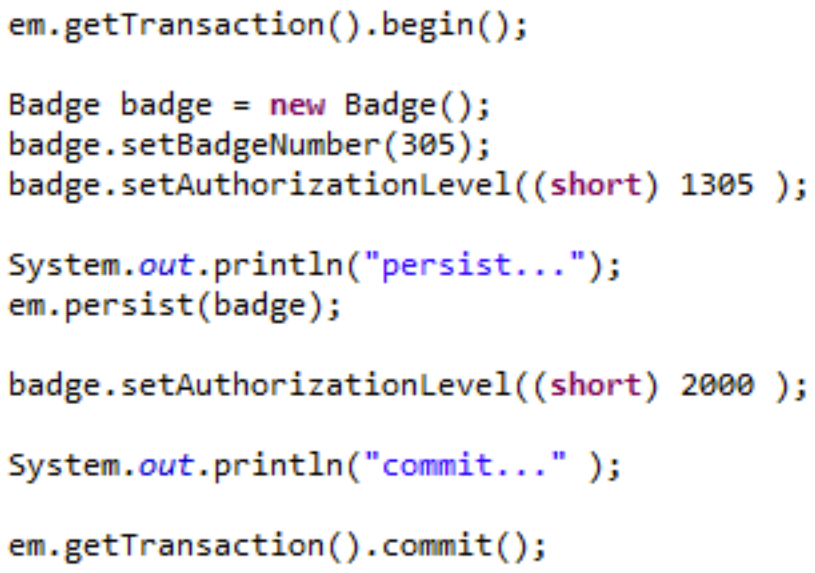
# ENTITYMANAGER: PERSIST(E)

- Managing instances with ”persist()”
  - The PersistenceContext contains a reference on the instance passed to it as a parameter.
  - Any later changes to this instance will therefore be implicitly recorded by the PersistenceContext and reflected in the database during the INSERT
  - The call "em.contains(e)" returns true if e is a reference to the entity passed to "persist”
  - If the instance is already present in the context: no error
  - If another instance with the same primary key is already present in the context:
    - PersistanceException ( NonUniqueObjectException )

# ENTITYMANAGER: PERSIST(E)

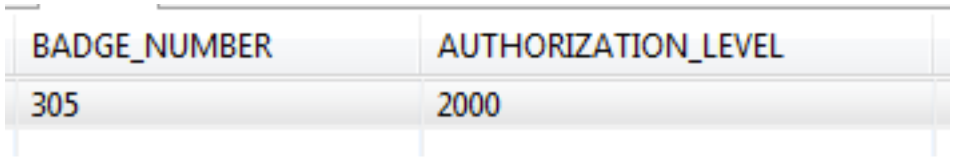


| | |
|---|---|
| void | **persist(Object entity)**<br>Make an instance managed and persistent. |



```
em.getTransaction().begin();

Badge badge = new Badge();
badge.setBadgeNumber(305);
badge.setAuthorizationLevel((short) 1305 );

System.out.println("persist...");
em.persist(badge);

badge.setAuthorizationLevel((short) 2000 );

System.out.println("commit..." );

em.getTransaction().commit();
```

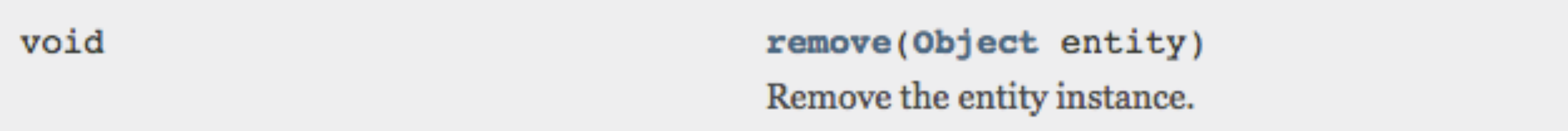
Then



| BADGE_NUMBER | AUTHORIZATION_LEVEL |
|---|---|
| 305 | 2000 |

The modification made after the call to ”persist()” is taken into account

# ENTITYMANAGER: REMOVE(E)

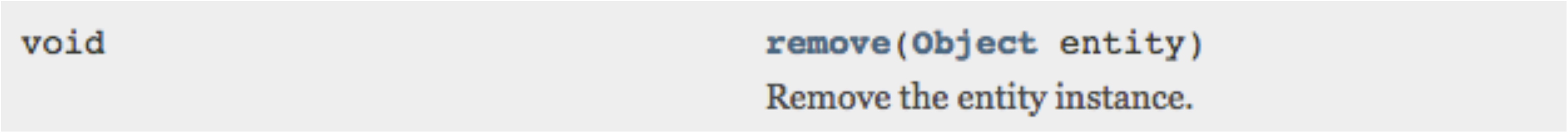


```
void                         remove(Object entity)
```

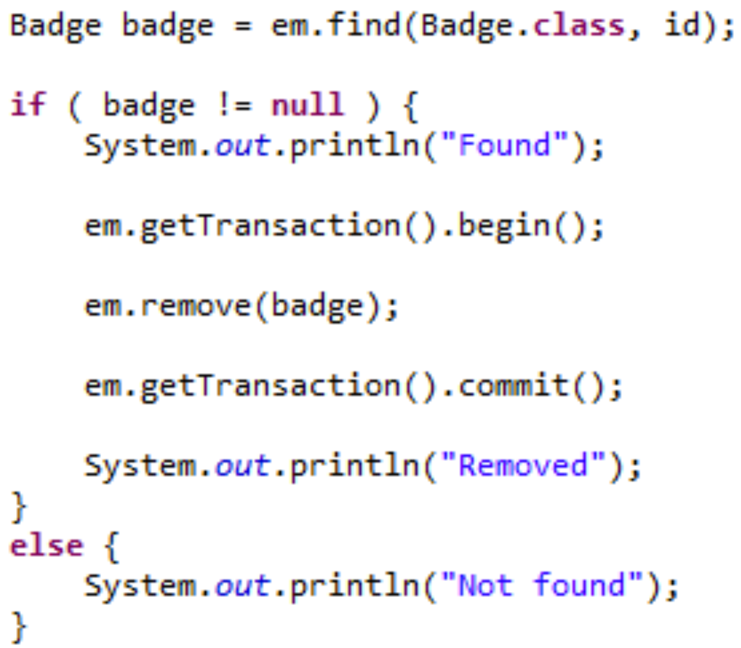
Remove the entity instance.

- Have to be used in an active transaction
- Comportment:
  - Change of state: “Removed”
    - If the state is “New” : Ignored
    - If state is “Managed” : Turns to “Removed”
    - If state is “Removed” : Ignored
    - If state is “Detached” : **IllegalArgumentException**

# ENTITYMANAGER: REMOVE(E)



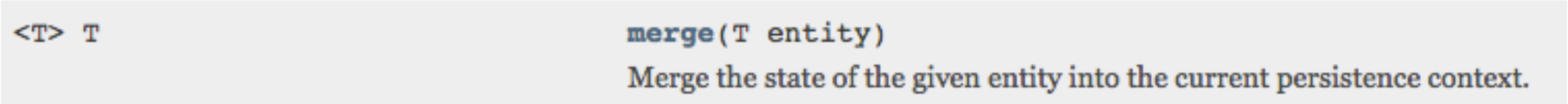
| | |
|---|---|
| void | **remove(Object entity)**<br>Remove the entity instance. |



```
Badge badge = em.find(Badge.class, id);

if ( badge != null ) {
    System.out.println("Found");

    em.getTransaction().begin();

    em.remove(badge);

    em.getTransaction().commit();

    System.out.println("Removed");
}
else {
    System.out.println("Not found");
}
```

You can use remove() on a managed Entity only

# ENTITYMANAGER: PERSIST(E)

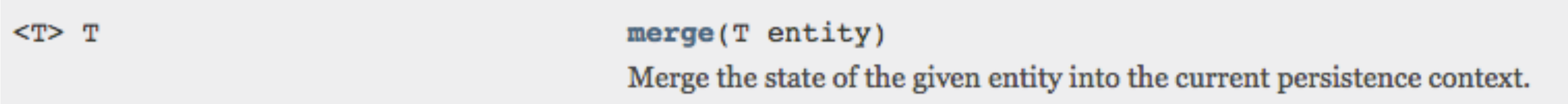


```
<T> T          merge(T entity)
```

Merge the state of the given entity into the current persistence context.

- Have to be used in an active transaction
- Work by copy
- Comportment:
  - Change of state: “Managed”
    - If the state is “New” : Create a new entity and copy it
    - If state is “Detached” : Copy in the existing entity
    - If state is “Managed” : Ignored
    - If state is “Removed” : **IllegalArgumentException**

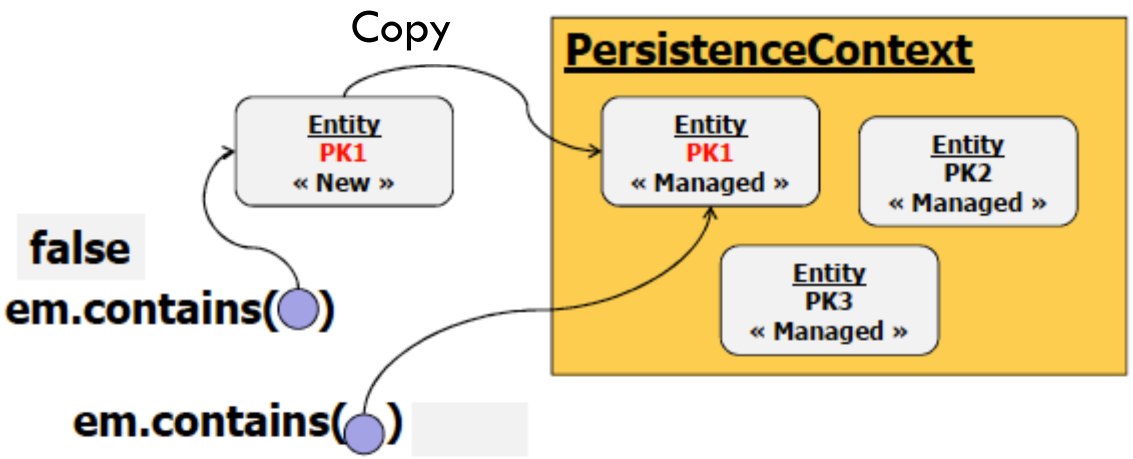
# ENTITYMANAGER: MERGE(E)



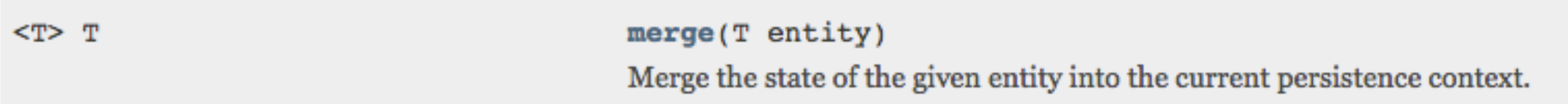
```
<T> T                          merge(T entity)
```

Merge the state of the given entity into the current persistence context.

- Merge = Fusion of 2 entities: Copying an instance into another one.



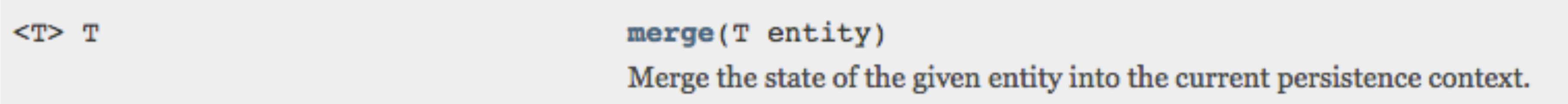
# ENTITYMANAGER: MERGE(E)



| | |
|---|---|
| `<T> T` | `merge(T entity)`<br>Merge the state of the given entity into the current persistence context. |

- Merge = Fusion of 2 entities: Copying an instance into another one.
  - The entity passed as a parameter is copied into another instance in the persistence context
  - If the entity does not exist in the context: It get loaded from the DB or a new instance is created.
  - Therefore there is 2 distinct instances:
    - « em.contains(e) » returns false ( “e” refer to a different entity)
    - Further modifications will have no effects on “e”

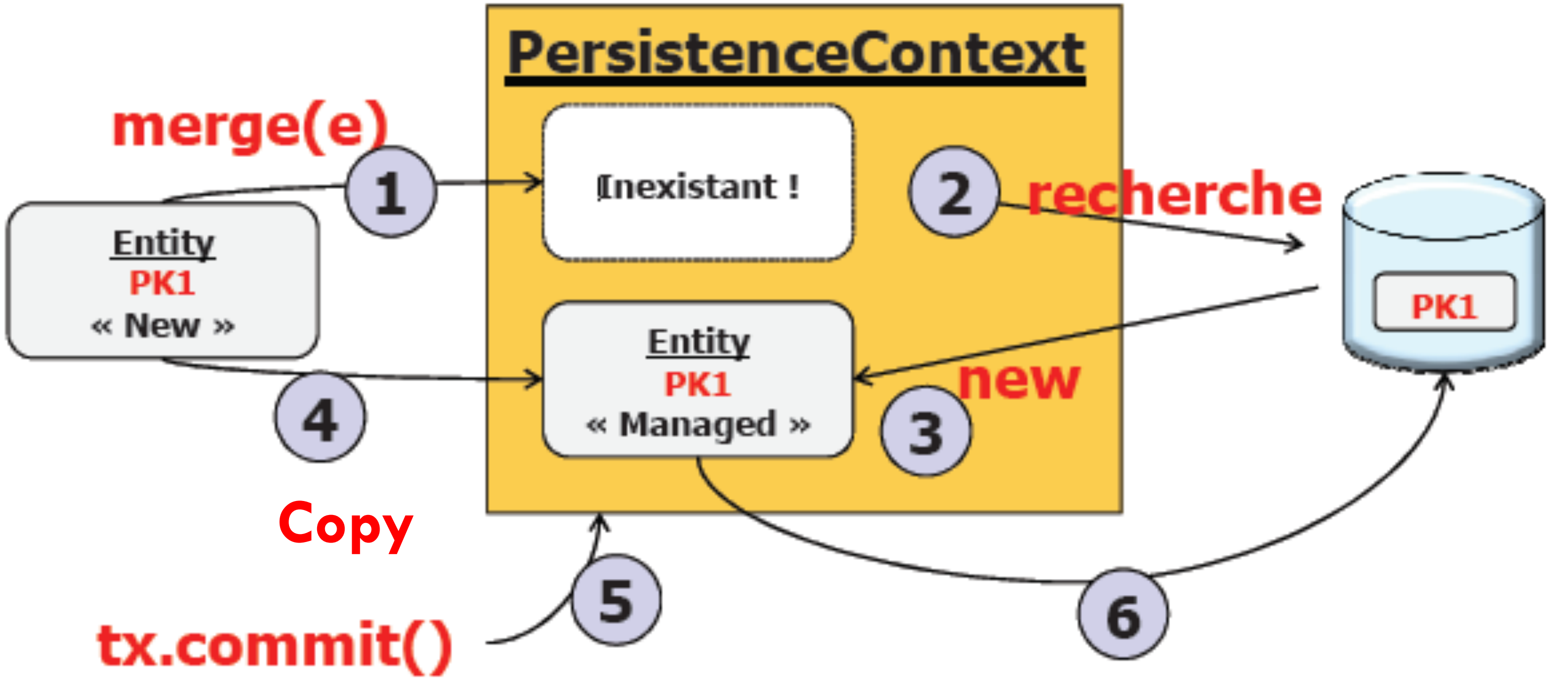
# ENTITYMANAGER: MERGE(E)



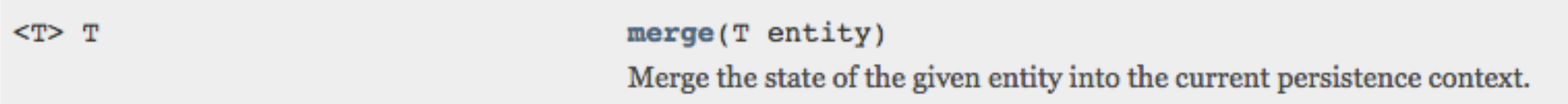
```
<T> T                    merge(T entity)
```

Merge the state of the given entity into the current persistence context.

- Example

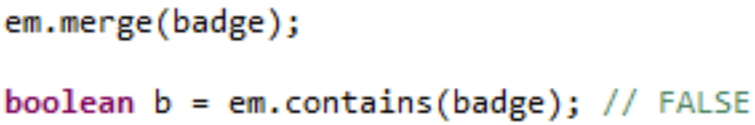


# ENTITYMANAGER: MERGE(E)

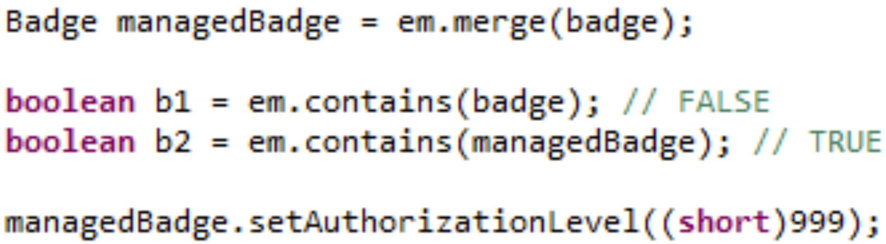


| | |
|---|---|
| `<T> T` | `merge(T entity)`<br>Merge the state of the given entity into the current persistence context. |

- Examples



```
em.merge(badge);

boolean b = em.contains(badge); // FALSE
```



```
Badge managedBadge = em.merge(badge);

boolean b1 = em.contains(badge); // FALSE
boolean b2 = em.contains(managedBadge); // TRUE

managedBadge.setAuthorizationLevel((short)999);
```

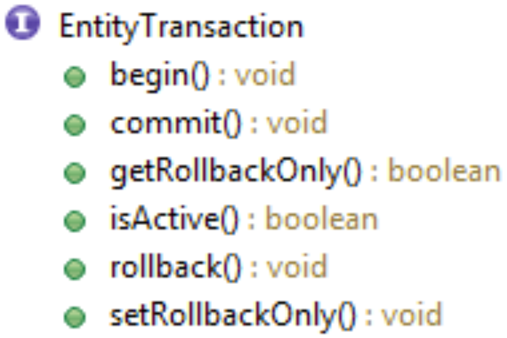
# TRANSACTION MANAGMENT

# TRANSACTION IN A NUTSHELL

- A transaction is a group of SQL instructions performing atomic functional processing:
  - A transaction have to be: **A**tomic, **C**onsistent, **I**solated and **D**urable (ACID)
    - Atomic: Indivisible.
    - Consistent: The final content have to be coherent in the database.
    - Isolated: When 2 transactions are executed at the same time, they should not interfere each others.
    - Durable: The final result of a transaction is conserved.

# USING TRANSACTIONS

- In JPA, transaction management happen through the "**EntityTransaction**" interface



EntityTransaction
- begin() : void
- commit() : void
- getRollbackOnly() : boolean
- isActive() : boolean
- rollback() : void
- setRollbackOnly() : void

- A transaction instance is retrieved from the EntityManager

```
EntityTransaction transaction = em.getTransaction();
```

# TRANSACTION: USAGE

- Start

```
em.getTransaction().begin();
```

- Ending

```
em.getTransaction().commit(); // or
em.getTransaction().rollback();
```

- Exemple

```
em.getTransaction().begin();
remove(badge);
em.getTransaction().commit();
```

# TRANSACTION

JPQL AND SQL

# REQUEST: DIFFERENT LANGUAGES

- One of the main goals of JPA is to avoid using SQL to access database.
- JPA provide a query language independent of the database SQL (sometime specific): **Java Persistence Query Language (JPQL).**
- JPA also provides a Java API to build queries dynamically from Java method calls: **the "Criteria API"**
- Finally, queries expressed in native SQL are sometimes necessary: JPA can call written queries in SQL

# REQUEST: JPQL

- JPQL is a revision (extension and improvement) of EJBQL (Query Language for EJB
- The syntax remains very close to SQL(SELECT, FROM, WHERE, … )
- Main difference:
  - in JPQL we do not do a SELECT on a TABLE but on a JAVA CLASS (a type of entity)

```
SELECT e.name FROM Employee e
```

```
SELECT e FROM Employee e
WHERE e.department.name = ‘AB’
AND e.address.state IN (‘NY’, ‘CA’)
```

# REQUEST: JPQL

- The parameters of JPQL queries can be represented by a symbolic name preceded by ':'

```
SELECT b FROM Badge b
WHERE b.badgeNumber >= :min
AND b.badgeNumber <= :max
```

- Or by a number preceded by ’?'

```
SELECT b FROM Badge b
WHERE b.badgeNumber >= ?1
AND b.badgeNumber <= ?2
```

# REQUEST: JPQL – USAGE WITH JAVA

- Request without parameter

```
    final String QUERY = "SELECT b.badgeNumber FROM Badge b ";
    Query query = em.createQuery(QUERY);
    //--- Execute query
    System.out.println("execute query ...");
    List<Integer> list = query.getResultList();System.out.println("Number
of badges : "+list.size());for(
    Integer i:list)
    {
        System.out.println(" . badge number : " + i);
    }
```

```
final String QUERY = "SELECT b FROM Badge b" ;
...
List<Badge> list = query.getResultList() ;
..
```