Non correspondance des modèles objet et relationnel

- **Définition de l'*Impedance mismatch***
- **Exemple simple de correspondance**
- **Identification des objets**
- **Traduction des associations**
- **Traduction de l'héritage**
- **Objet dépendant**
- **Navigation dans le graphe d'objets (types et stratégies de chargement)**

Bibliographie pour le mapping Objet/relationnel (1/2)

- [BK07] ***Java Persistence with Hibernate, Revised Edition of Hibernate in Action***, de Christian Bauer and Gavin King, Manning Publications, 2007 – nouvelle édition en nov. 2015.

- ***Hibernate Recipes 2015 : A Problem-Solution Approach***, de Gary Mak Srinivas Guruzu , et Joseph Ottinger, 2nd Revised edition, Springer-Verlag 2015

- [Fow03] ***Patterns of Entreprise Application Architecture***, de Martin Fowler, Addison Wesley, 2003

- ***Mapping Objects to Relational Databases: O/R Mapping In Detail*, de Scott W. Ambler, 2006**
  http://www.agiledata.org/essays/mappingObjects.html

- [Fus97] ***Foundations of Object-Relational Mapping*, de Mark L. Fussell, 1997**
  http://markfussell.emenar.com/blog/object-relational/

Bibliographie pour le mapping Objet/relationnel (2/2)

- *Data Mapper* - Martin Fowler
  https://www.martinfowler.com/eaaCatalog/dataMapper.html

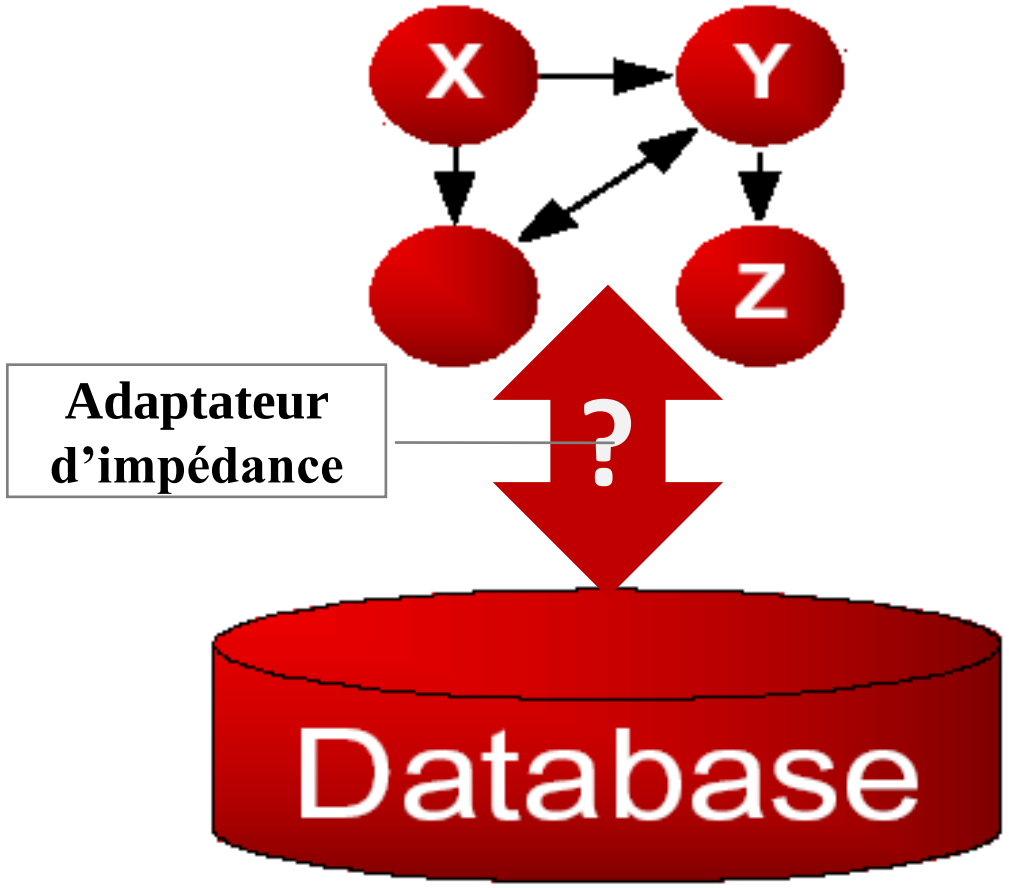
- Eclipse JNoSQL : une implémentation de référence de Jakarta NoSQL
  https://www.programmez.com/actualites/eclipse-jnosql-une-implementation-de-reference-de-jakarta-nosql-29541

- *What is JPA? Introduction to the Java Persistence API*
  https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html

- *Eclipse JNoSQL: One API to many NoSQL databases*
  https://www.eclipse.org/community/eclipse_newsletter/2018/april/jnosql.php

- Object-DB Mappers : site avec des liens vers les différents outils de mapping SQL et NoSQL
  https://people.cs.kuleuven.be/~dimitri.vanlanduyt/ondm-survey/#surveyresults

Défaut d'impédance (d'*Impedance mismatch)*



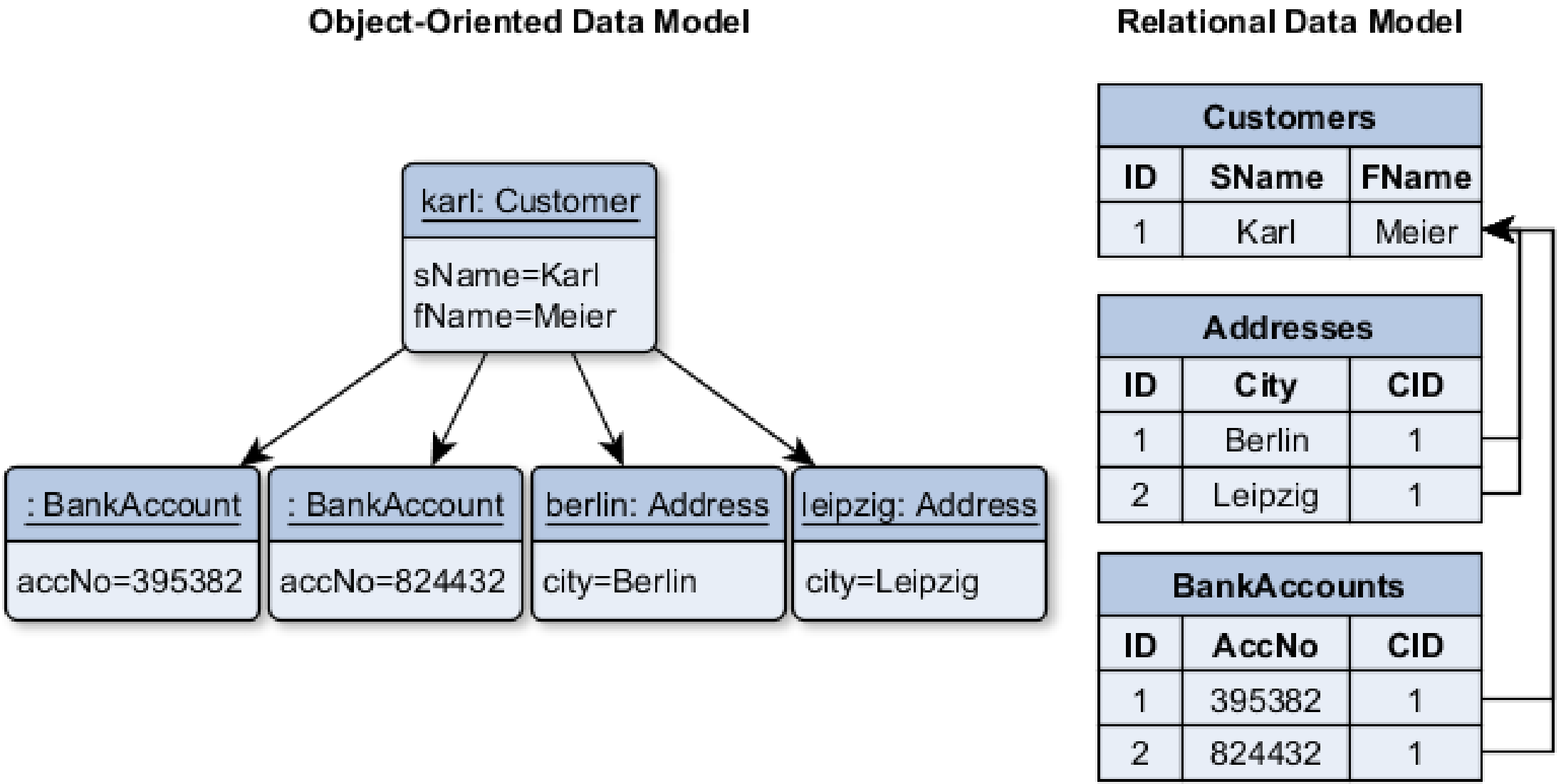
Propriétés à conserver :

- Objets complexes
- Identification des objets
- Encapsulation
- Classes
- Hiérarchie de classes
- Polymorphisme
- Navigation dans le graphe d'objets
- Cache objet

Propriétés à ajouter :

- Persistance
- Interrogation
- Gestion de la concurrence
- Sécurité et reprise après panne
- Gestion de la mémoire secondaire

Défaut d'impédance : exemple



Non correspondance des modèles objet et relationnel : quels problèmes ?

Impedance mismatch

- Entrée : modèle objet
- Sortie : modèle relationnel
- Problèmes de correspondance :
  - Identification des objets
  - Traduction des associations
  - Traduction de l'héritage
  - Navigation entre les objets
  - Objets dépendants

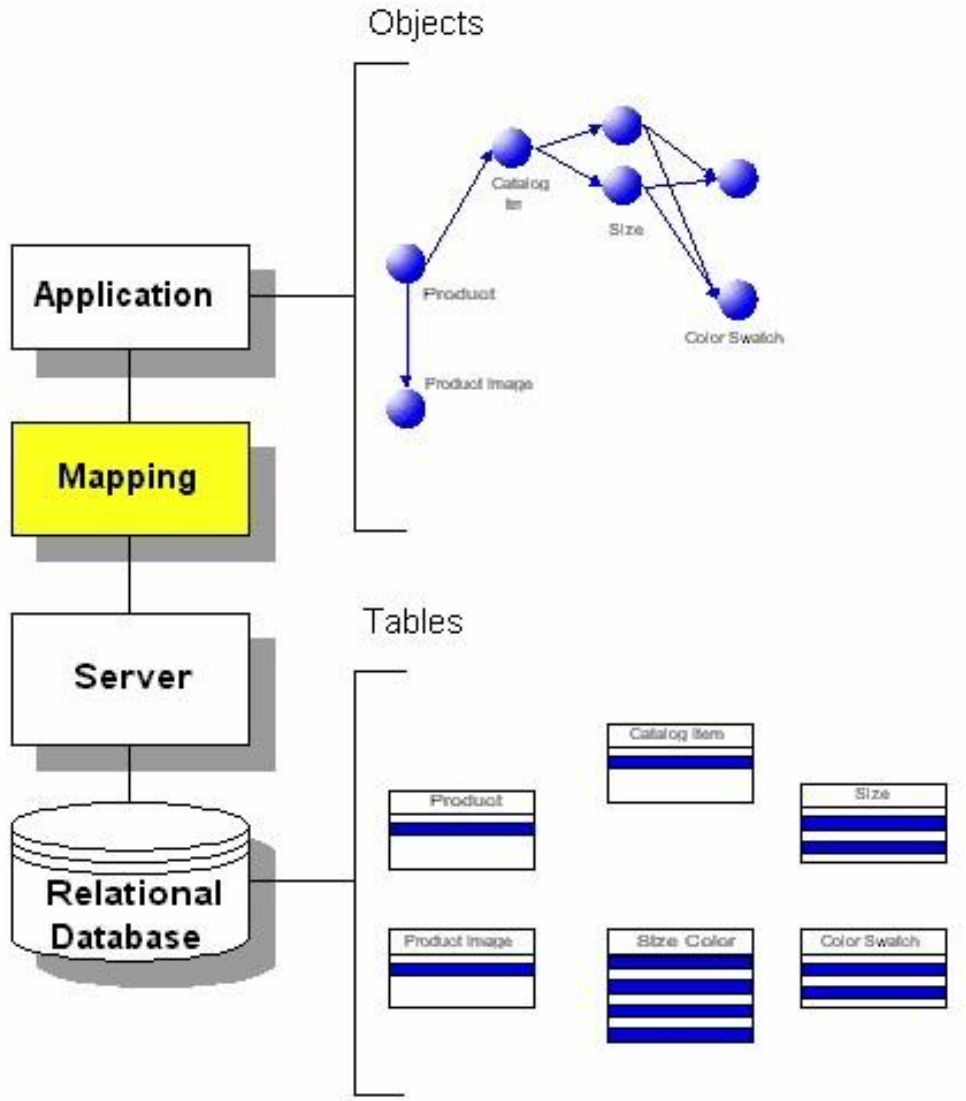


Figure reprise de http://www.service-architecture.com/object-oriented-databases/articles/impedance_mismatch.html

ORM : *Object/Relational Mapping*

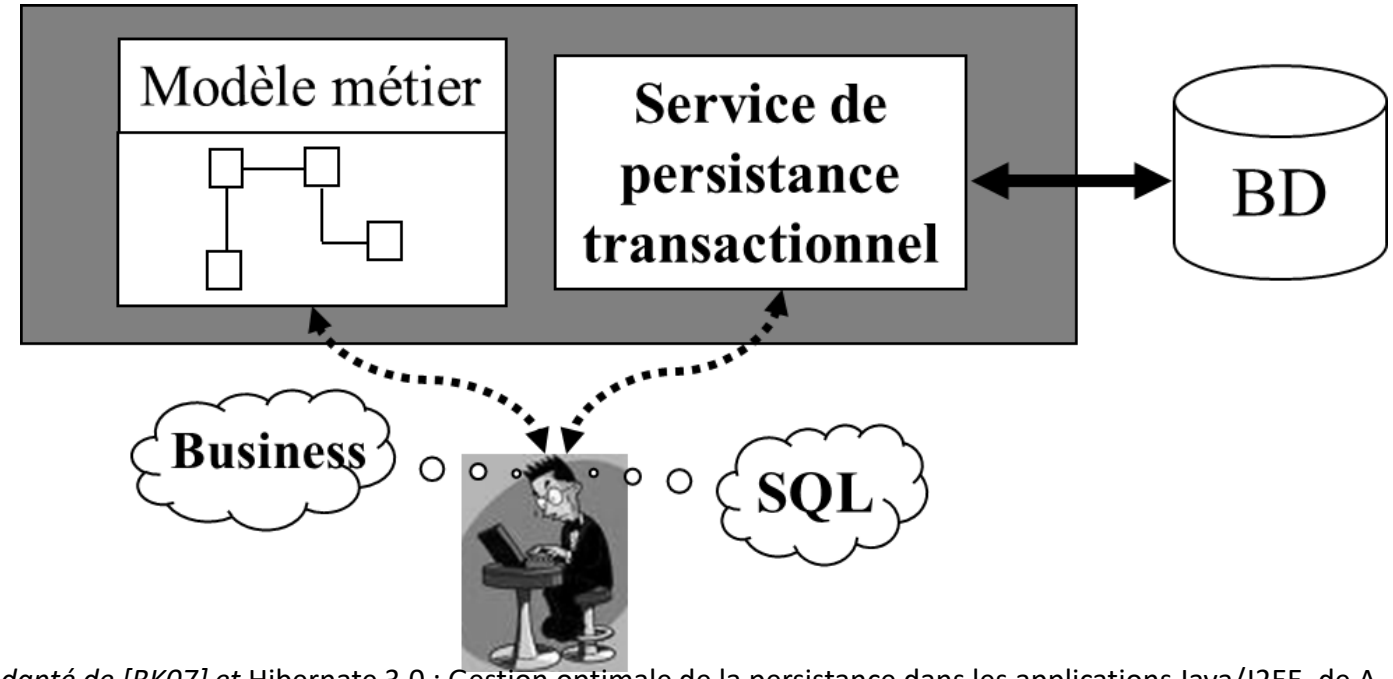
- **Persistance automatisée et « transparente » d'objets métiers vers une bases de données relationnelles** [BK05]
- Description à l'aide de **métadonnées** de la **transformation <u>réversible</u>** entre un modèle relationnel et un modèle de classes [BK05]
- Capacité à manipuler des données stockées dans une base de données relationnelles à l'aide d'un langage de programmation orientée-objet
- Techniques de programmation permettant de lier les bases de données relationnelles aux concepts de la programmation OO pour créer une "base de données orientées-objet virtuelle" [Wikipedia]

Couche d'accès au données
Couche de persistance

- Prise en charge de toutes les interactions entre l'application et la base de données
- Groupes de classes et de composants chargés du stockage et de la récupération des données [BK05]
- Possibilité de servir de cache pour les objets récupérés dans la base de données pour améliorer les performances
- Inclus un modèle (de métadonnées) des entités du domaine métier

Couche de persistance : à la charge du développeur (1/3)

- Possibilité de programmer manuellement une couche de persistance avec SQL/JDBC (*Java DataBase Connectivity*)
- Possibilité de masquer le JDBC complexe et le SQL non portable à la logique métier par un modèle de conception (ex. *active record*)



Couche de persistance : à la charge du développeur (2/3)

2 niveaux de qualité [Fus97] :

- **Relationnel pur** :
  - Application entièrement conçue autour du modèle relationnel et des opérations relationnelles réalisées en SQL
  - Modèle utilisé dans le cas d'applications simples, sans nécessité de réutilisation de code
  - Utilisation d'*Embedded SQL* ou *SQLJ* et de procédures stockées ⇒ décharge d'une partie du travail de la couche métier vers la base de données

Manque de portabilité et de maintenance à long terme

- **Correspondance objet légère (*Light Object Mapping*)** :
  - Correspondance codée manuellement entre les classes et les relations de la base de données
  - Masquage du SQL/JDBC programmé manuellement par l'utilisation de modèles de conception (*design pattern*) connu – ex. *active record*
  - Utilisée pour des applications ayant un petit nombre d'entités
  - Utilisation de procédures stockées

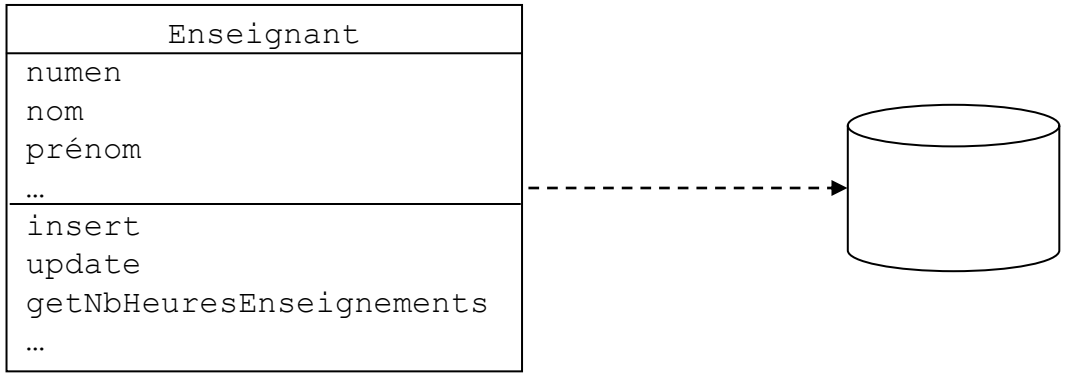
Couplage trop fort entre les classes métiers et le support de persistance utilisé

Couche de persistance : à la charge du développeur (3/3)

***Active record* (motif/patron de conception – *design pattern*)** :

- « *An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data* »
- Partie "Modèle" de l'architecture "Modèle Vue Contrôleur"
- Correspondance de chaque relation de la base avec la définition d'une classe : chaque colonne de la relation = une propriété de la classe
- Correspondance de chaque nuplet de la relation avec une instance de la classe correspondante :

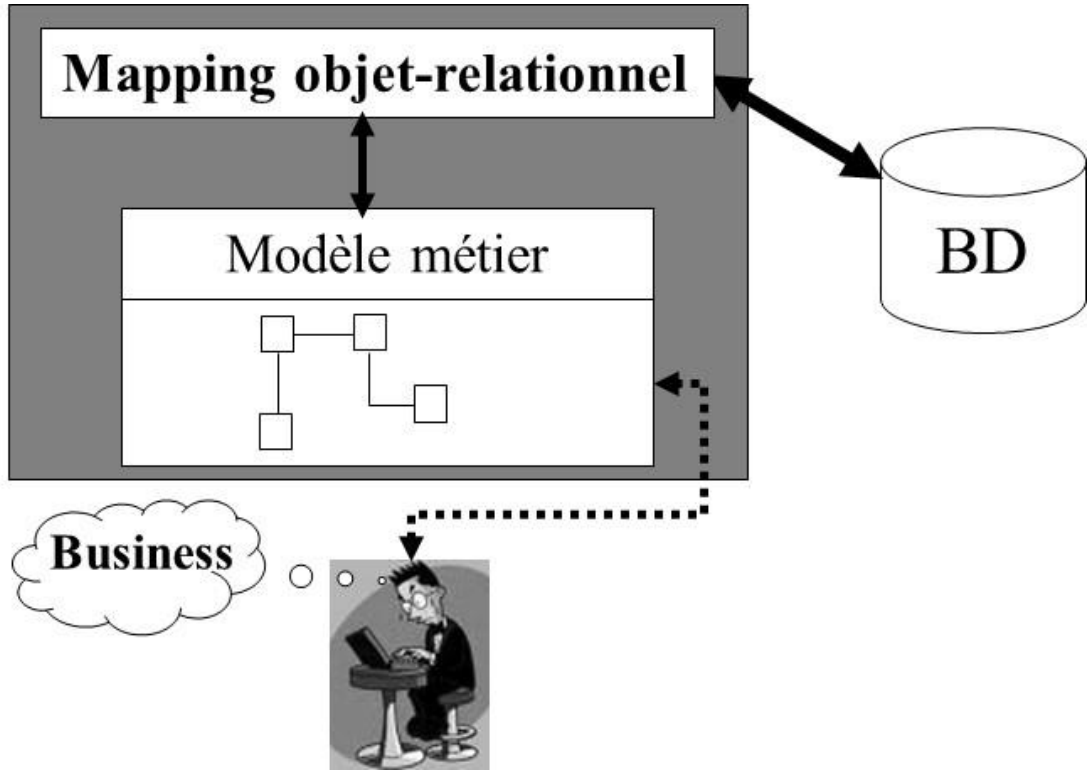
  création d'un nouvel objet ⇒ insertion d'un nouvel nuplet dans la relation
- Méthodes statiques de classes agissant sur l'ensemble des nuplets
- Requêtes CRUD (*Create*, *Read*, *Update*, *Delete*) pouvant être générées automatiquement



Couche de persistance : avec correspondance objet/relationnel (1/4)

- Utilisation de la couche de persistance comme un service rendant abstraite la représentation relationnelle indispensable au stockage final des objets
- Concentration du développeur sur les problématiques métier

Stratégie de persistance « transparente »



Couche de persistance : avec correspondance objet/relationnel (2/4)

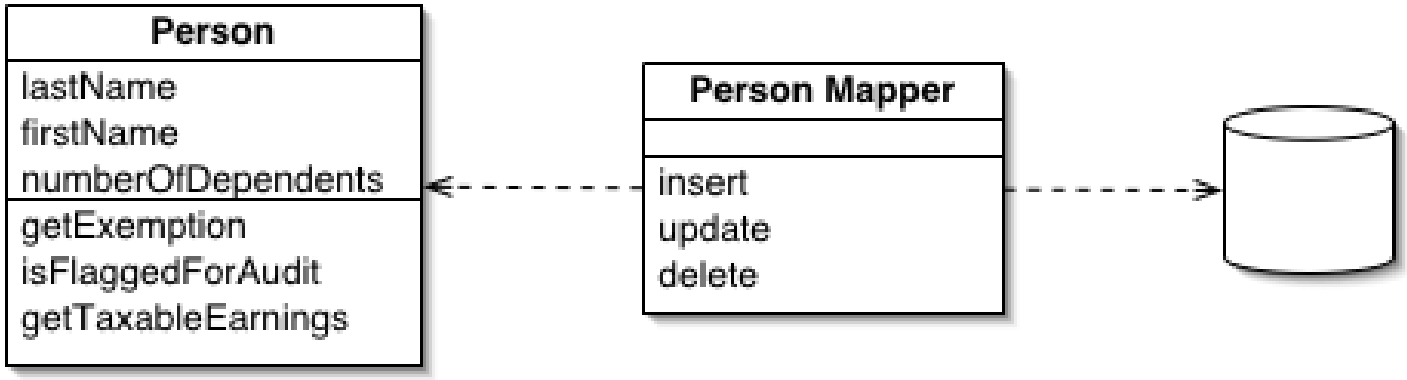
2 niveaux de qualité [Fus97] :

- **Correspondance objet moyenne (*Medium Object Mapping*)** :
  - Application conçue autour d'un modèle objet
  - SQL généré à la compilation par un outil de génération de code ou à l'exécution par le code de l'outil de correspondance (*mapping framework*)
  - Objets mis en cache par la couche de persistance
  - Pour des applications de taille moyenne incluant des transactions complexes
  - *Ex. EJB1.x/2.x Entity Beans*
- **Correspondance objet complète (*Full Object Mapping*)** :
  - Prise en compte de toutes les propriétés objets : composition, héritage, polymorphisme, persistance par accessibilité
  - Persistance *presque* transparente pour le développeur
  - Pas d'héritage d'une classe de base par les classes persistantes, ni d'interface spéciale
  - Stratégie d'extraction efficace (précoce ou tardive) et stratégies de mise en cache implémentées de manière transparente
  - *Ex. JPA 2.2 (Java Persistence API) in Jakarta EE*

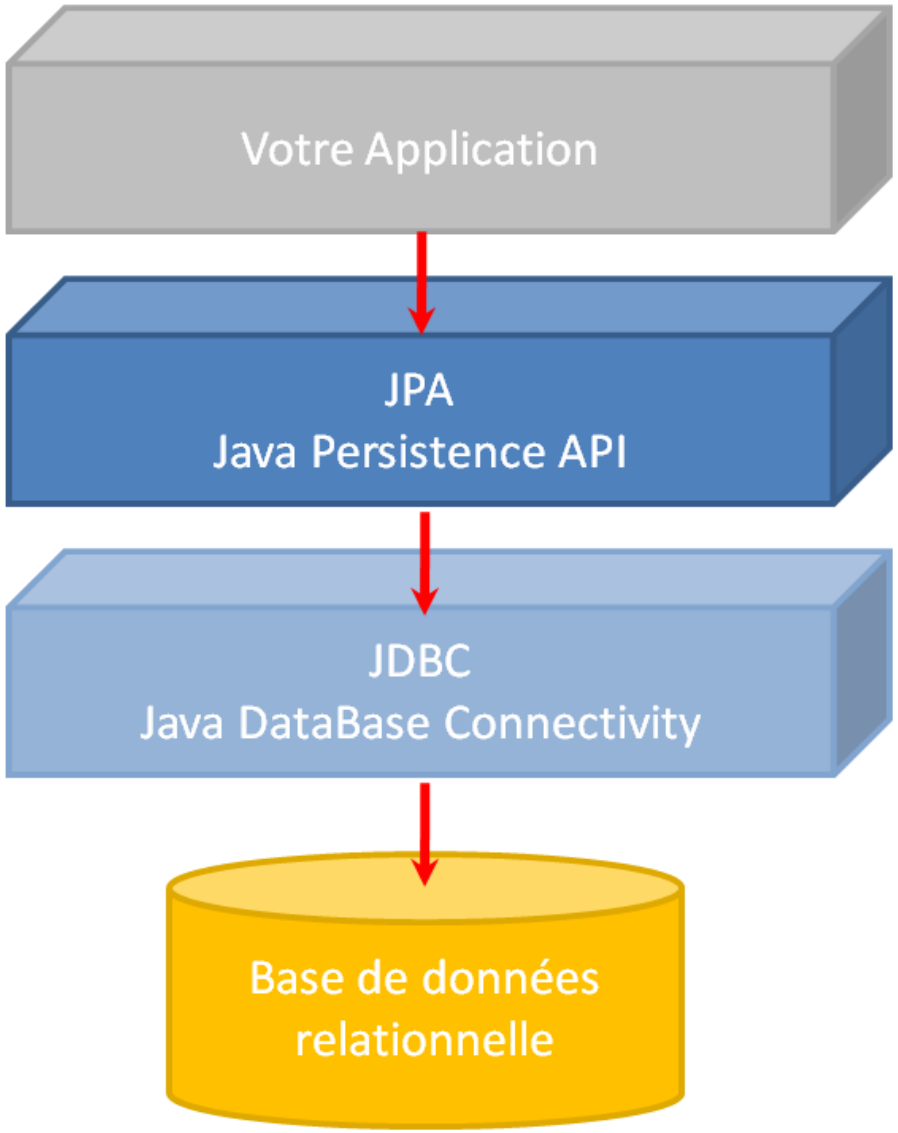
Couche de persistance : avec correspondance objet/relationnel (3/4)

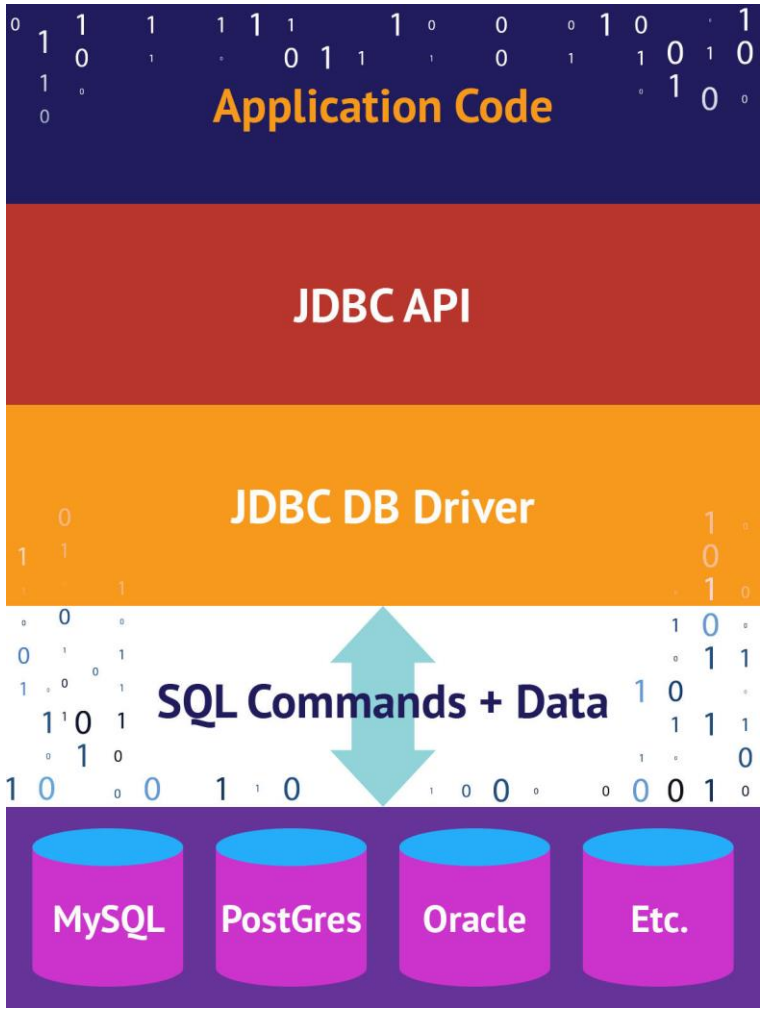
***Data Mapper* ou DAO (*Data Access Object*)** :

- « *A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.*»
- Requêtes CRUD (*Create*, *Read*, *Update*, *Delete*) pouvant être générées automatiquement
- Différence entre un DAO et un active record : « Les objets manipulant les données n'ont pas accès au code permettant de sauvegarder ces données dans la base »
- Utilité :
  - Faciliter la modification du modèle de base de données
  - Factoriser le code d'accès aux données
  - Faciliter l'optimisation des accès à la base en les regroupant au sein d'objets particuliers



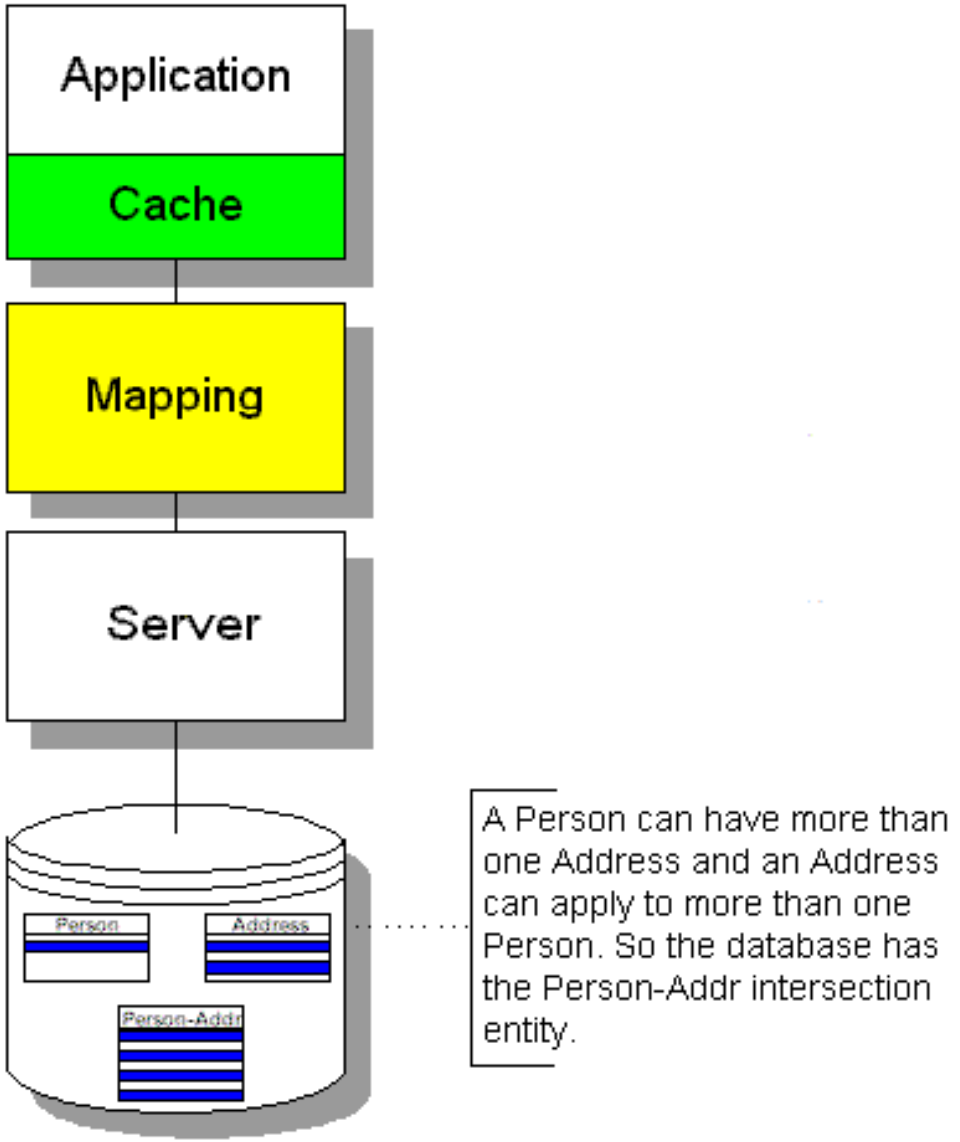
Couche de persistance : avec correspondance objet/relationnel (4/4)





Exemple de correspondance Objet/relationnel (1/15)

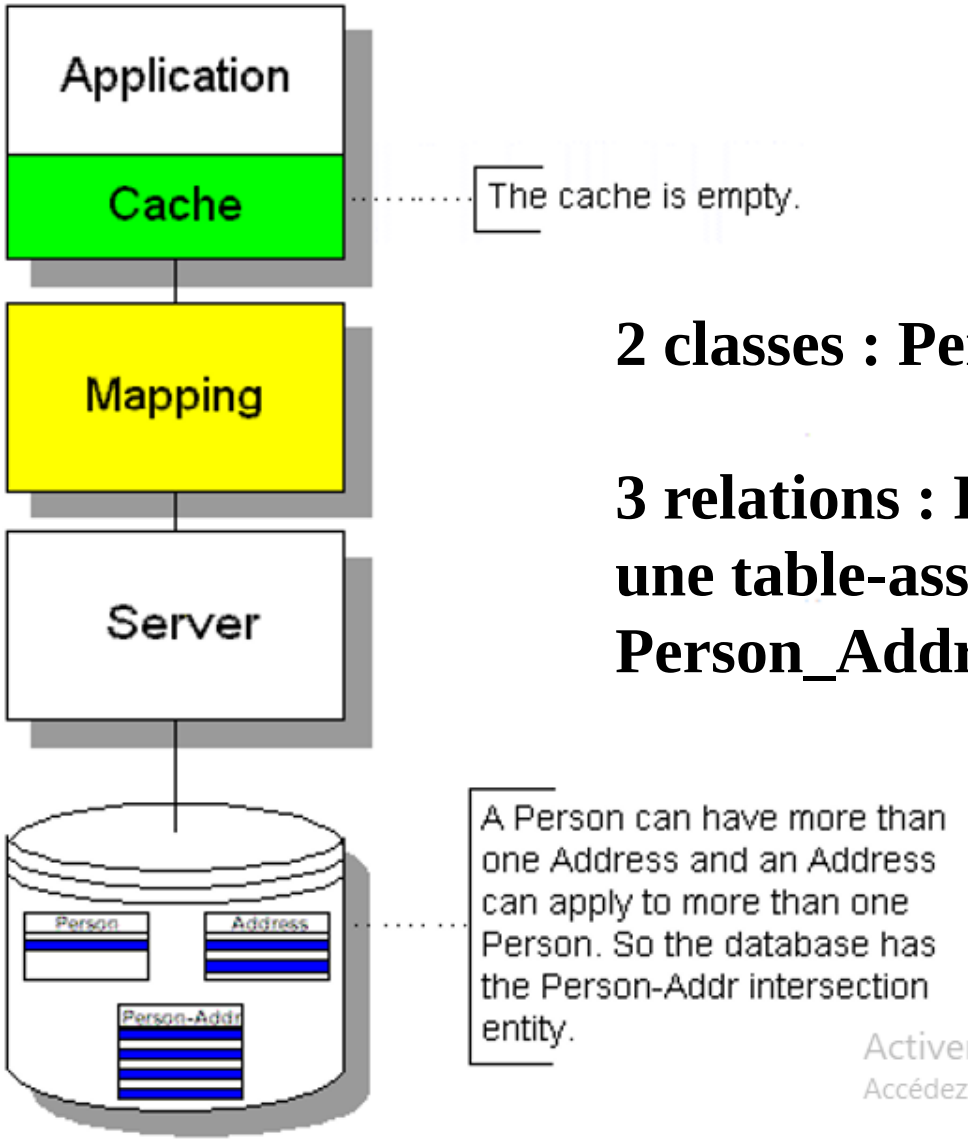
Persistance *presque* transparente



Repris de http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html

Exemple de correspondance Objet/relationnel

Persistance *presque* transparente



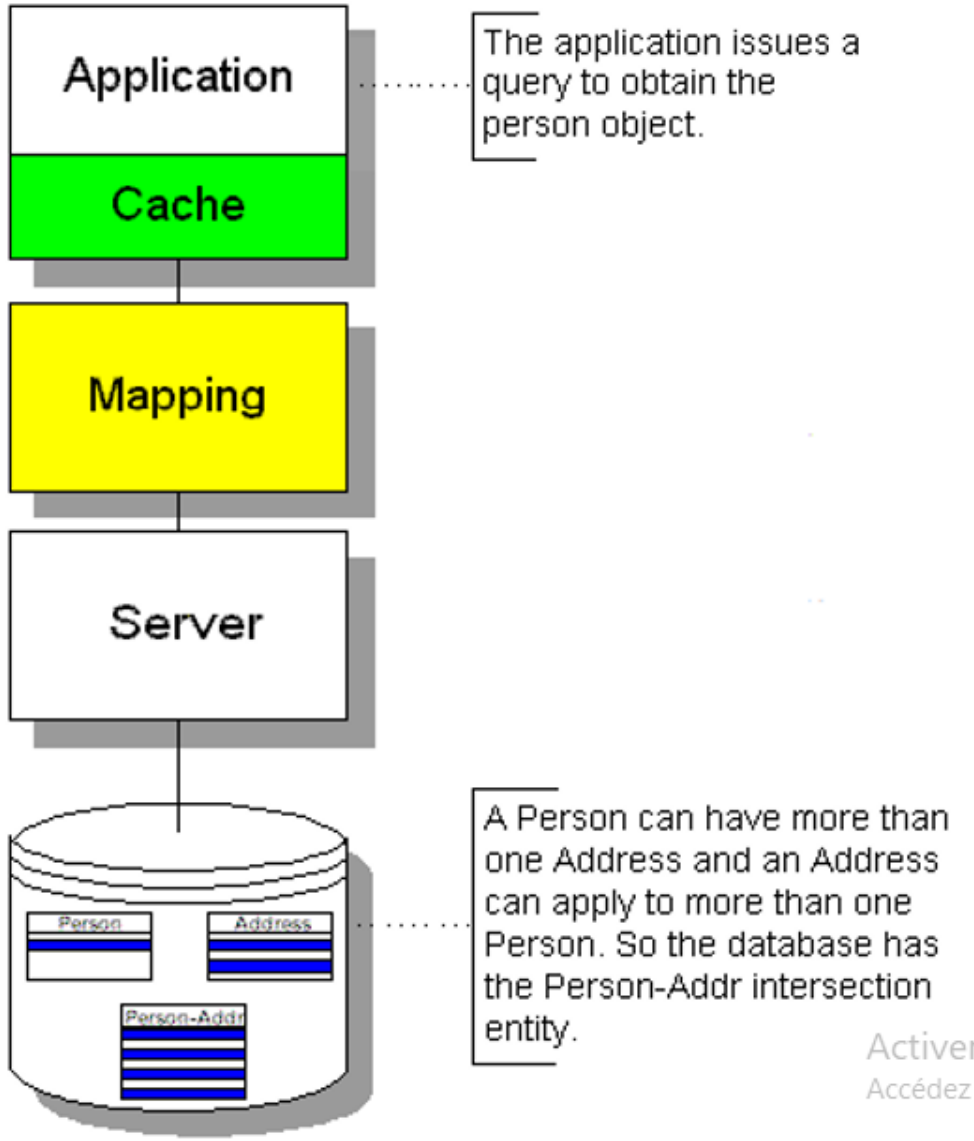
2 classes : Person et Address

3 relations : Person, Address et une table-association Person_Address

Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

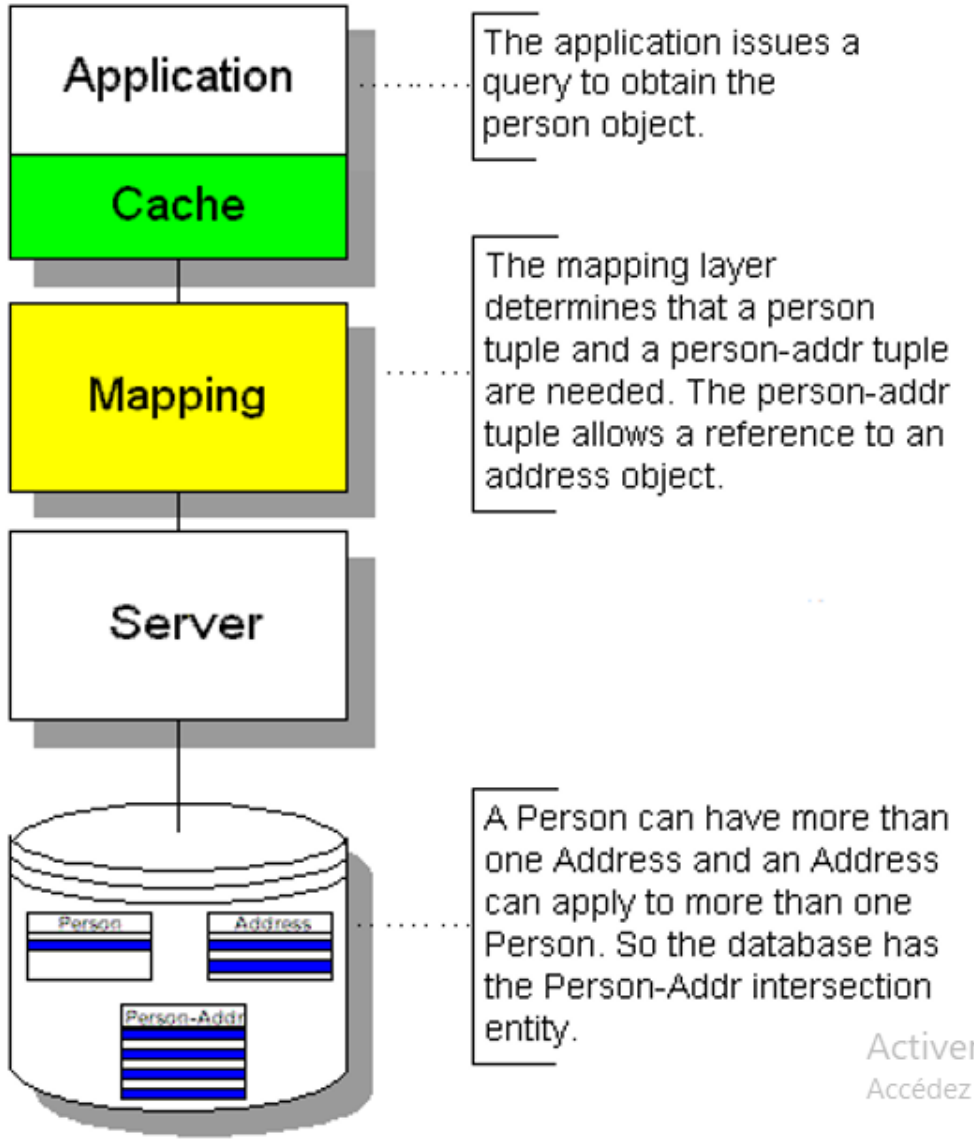
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

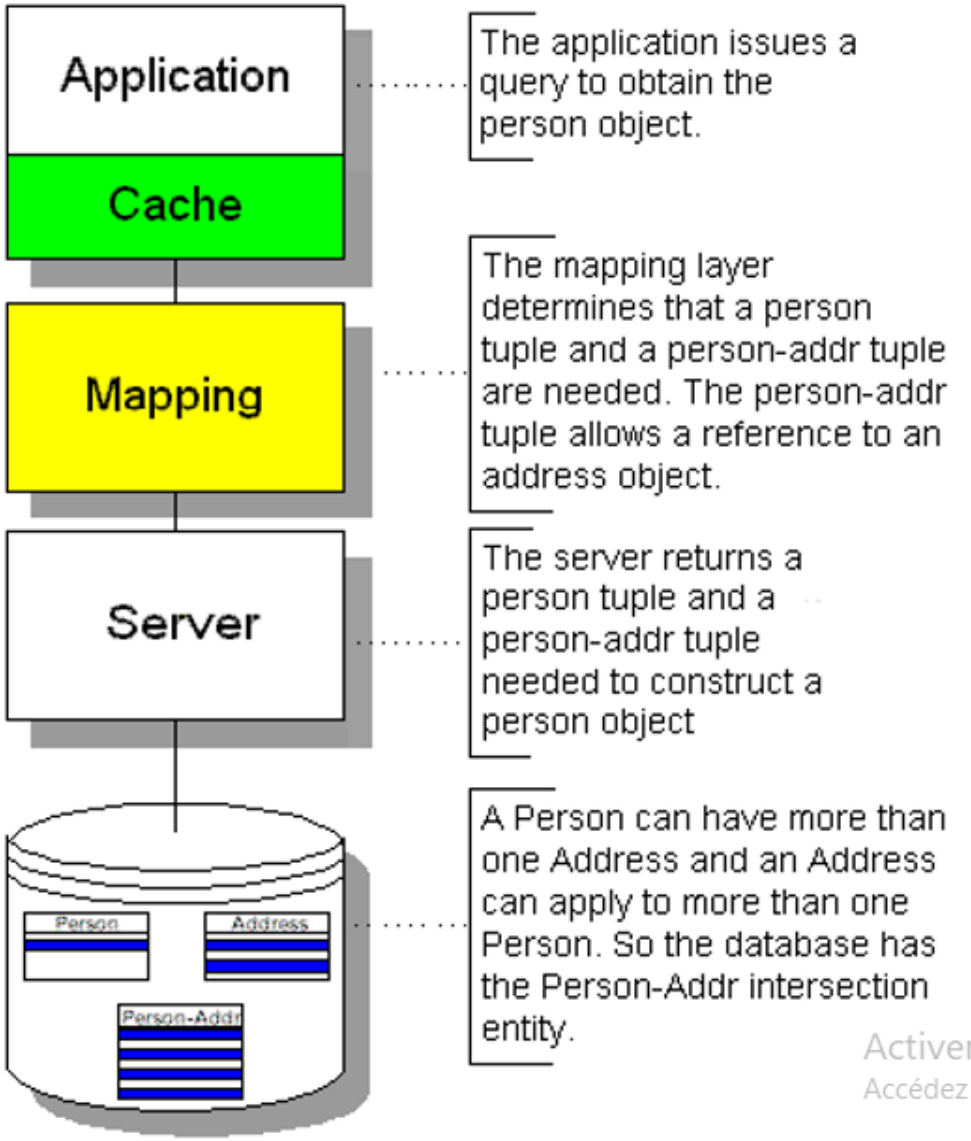
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

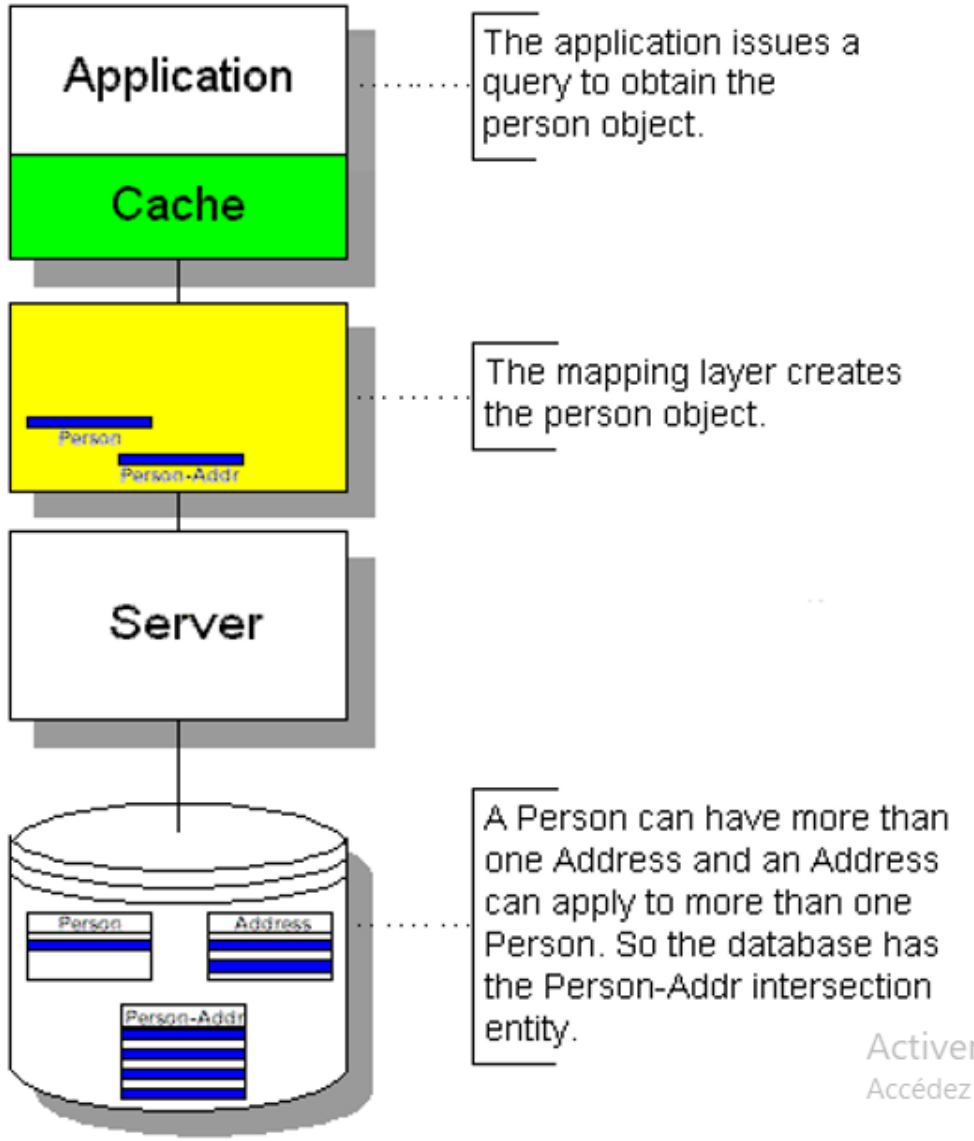
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

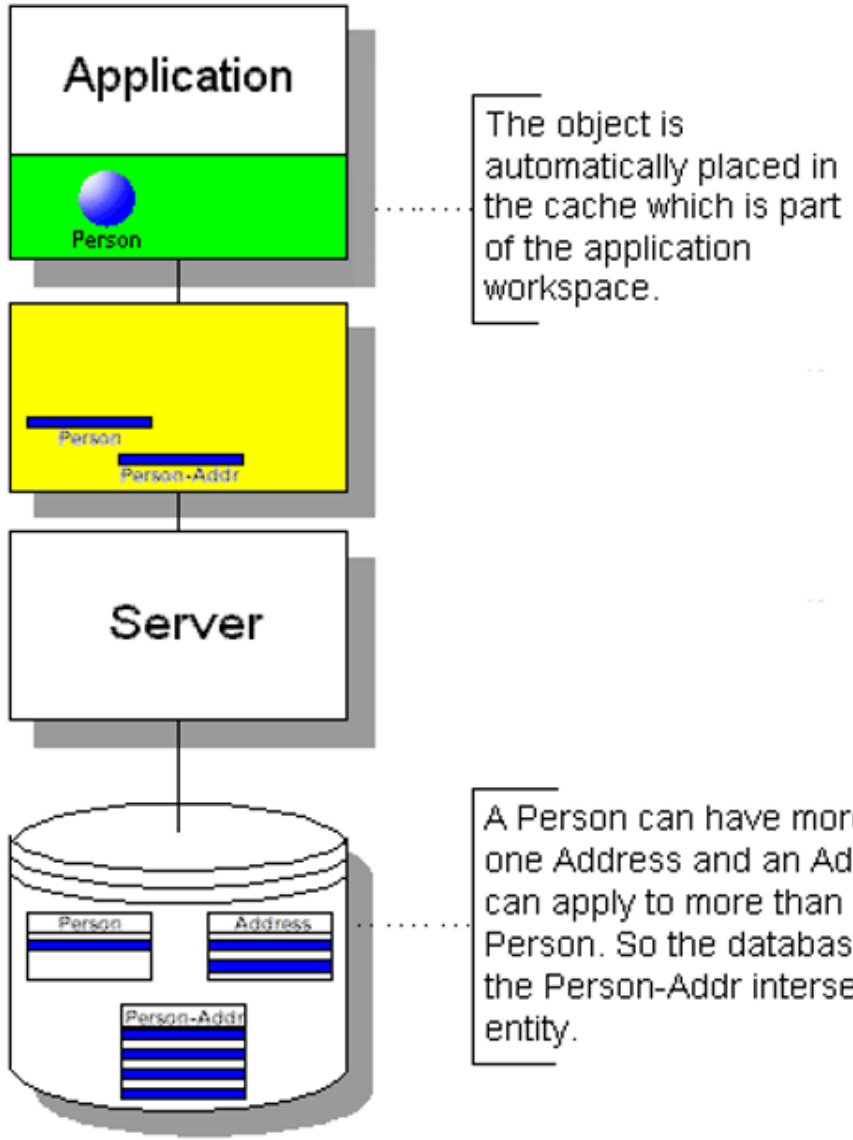
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

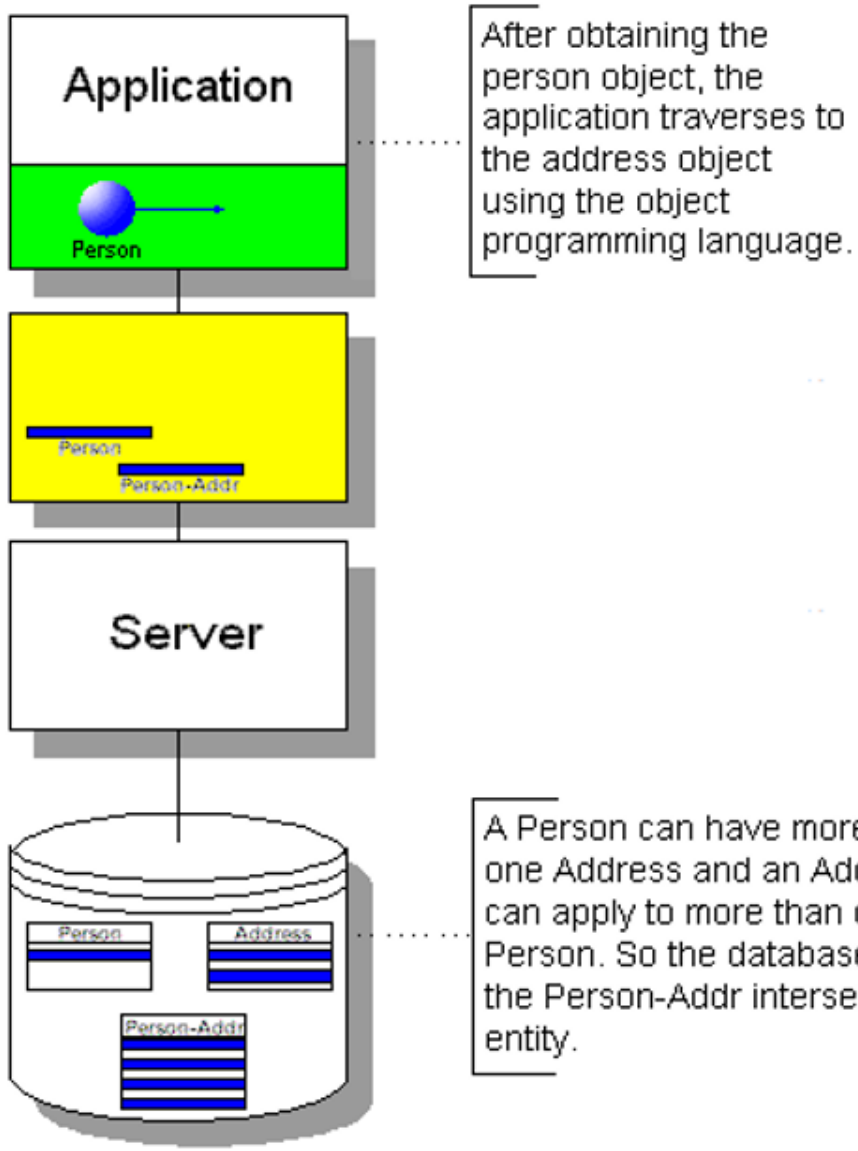
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

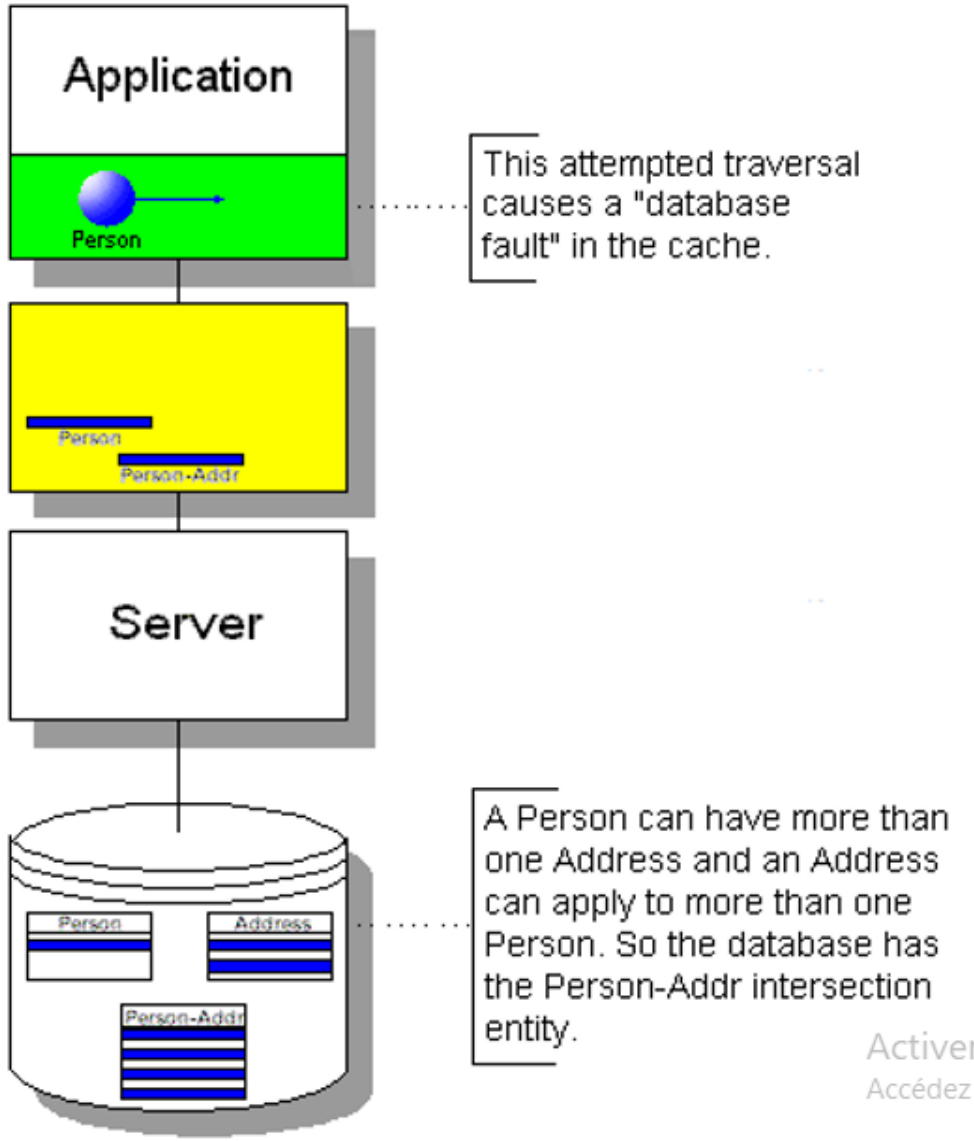
Persistance *presque* transparente



Repris de `http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html`

Exemple de correspondance Objet/relationnel

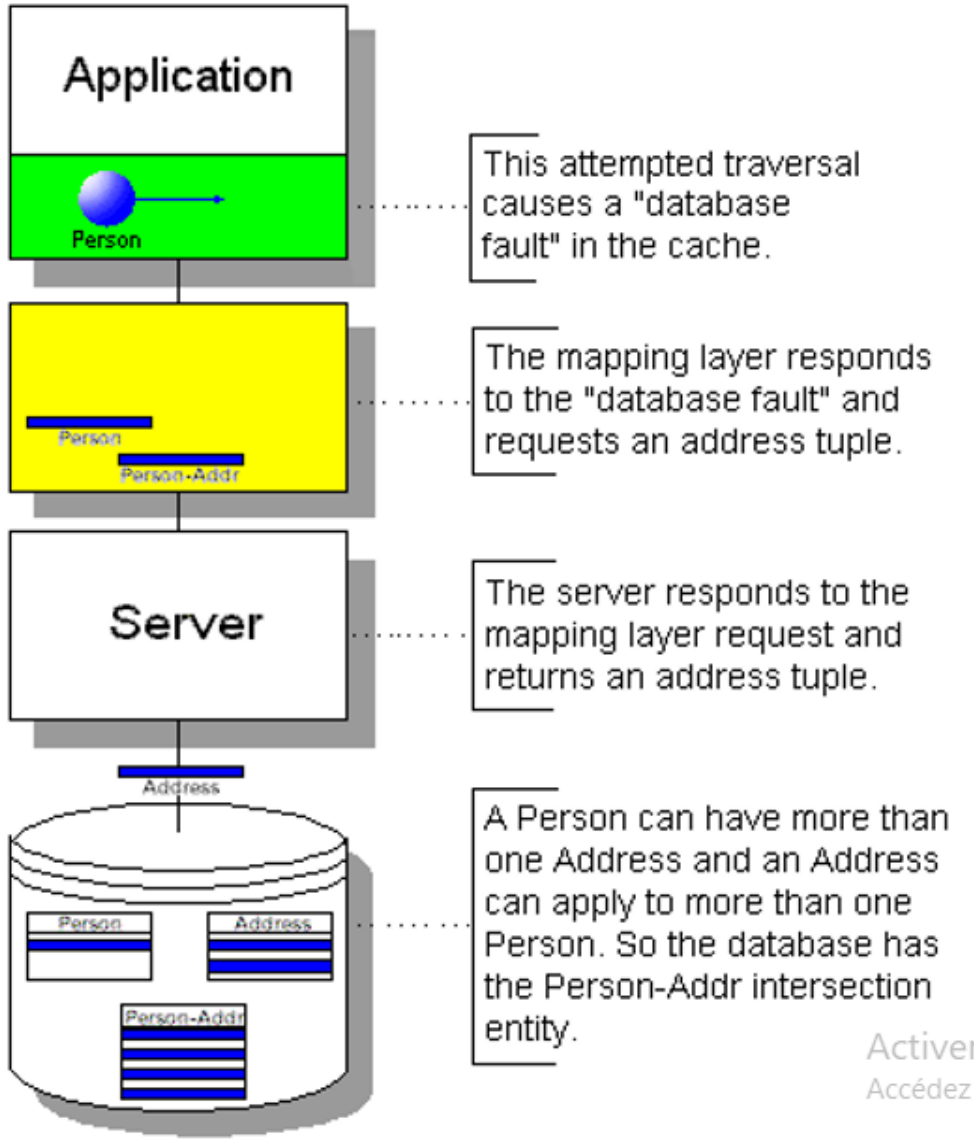
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

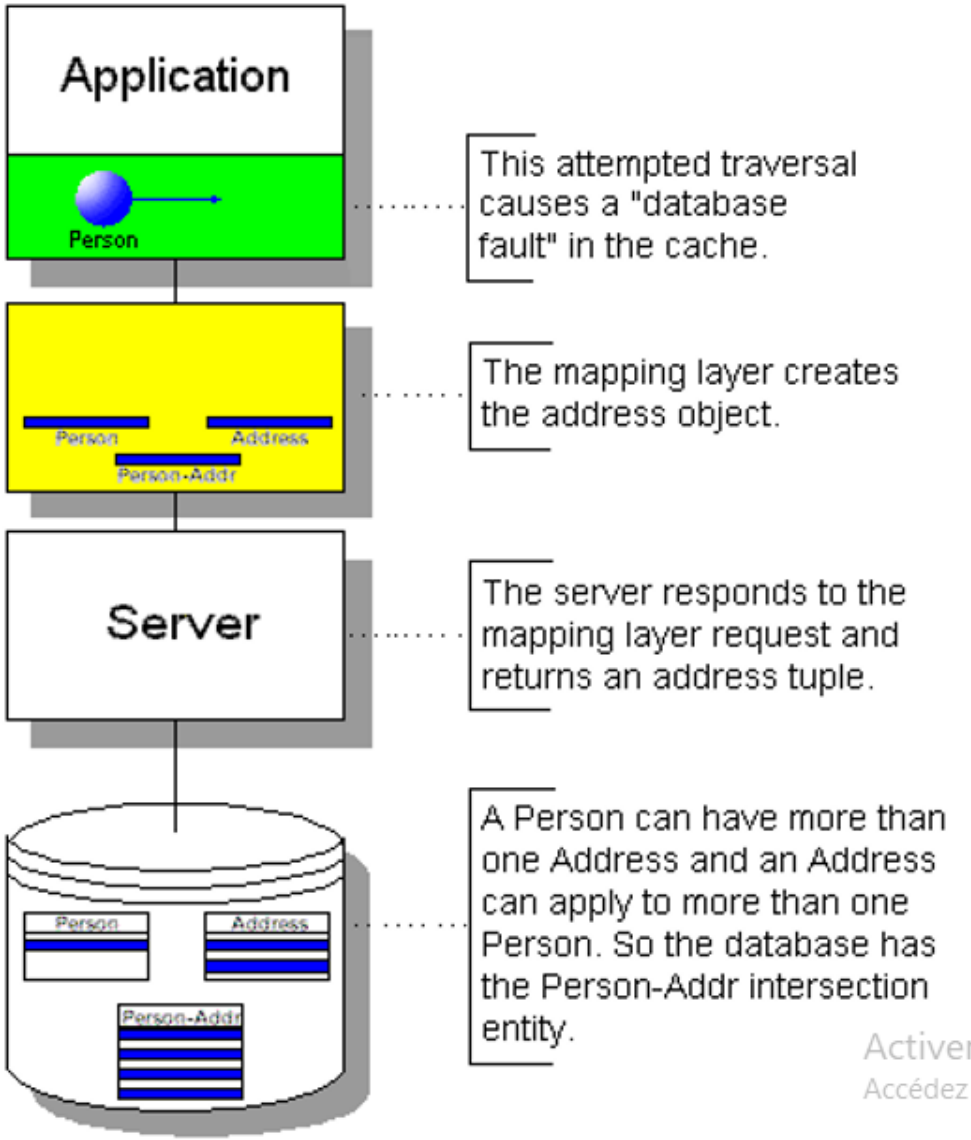
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

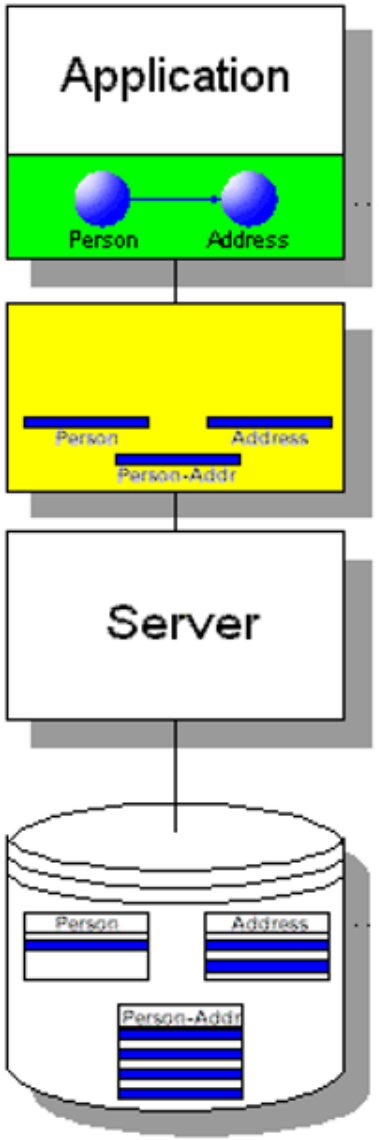
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

Persistance *presque* transparente



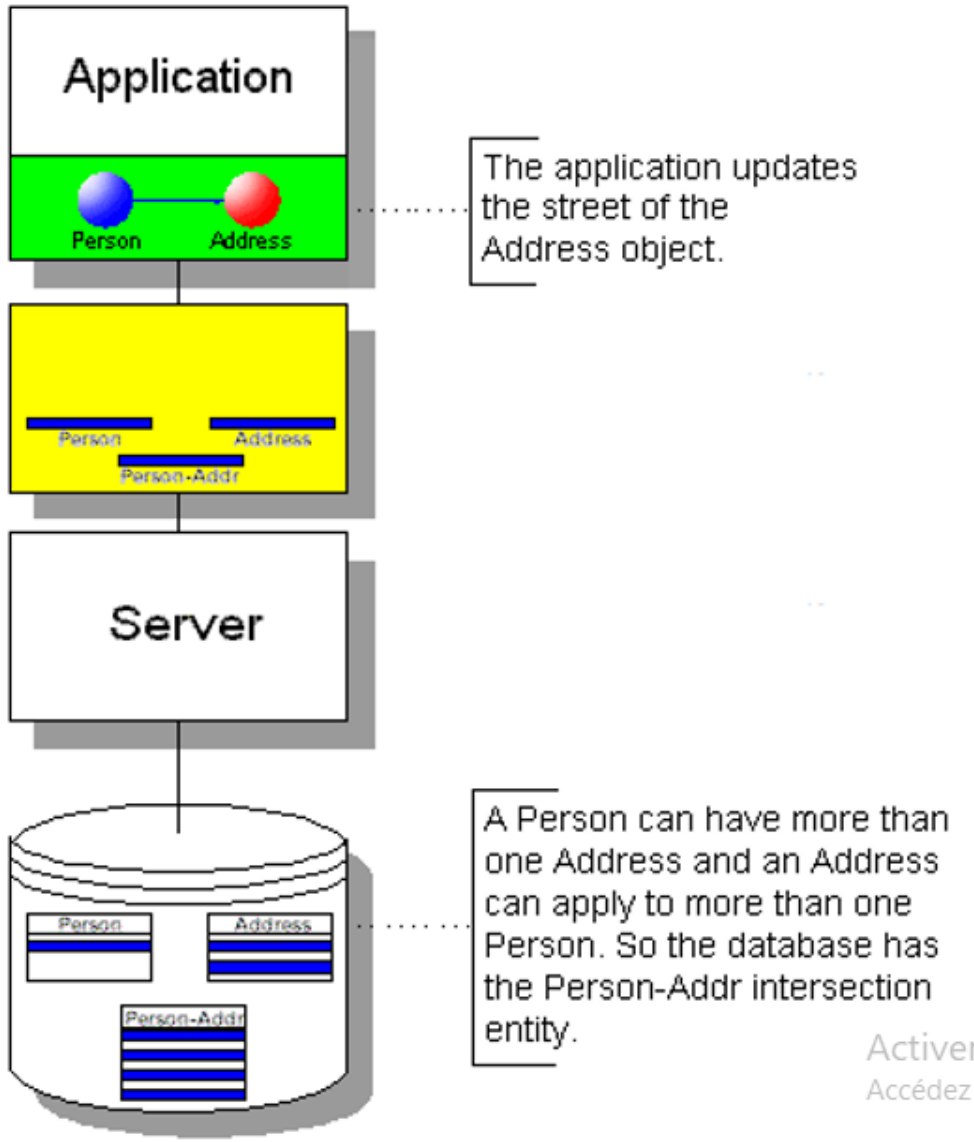
The application workspace now has the person object referencing the address object. The objects may be directly manipulated by the object programming language.

A Person can have more than one Address and an Address can apply to more than one Person. So the database has the Person-Addr intersection entity.

Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

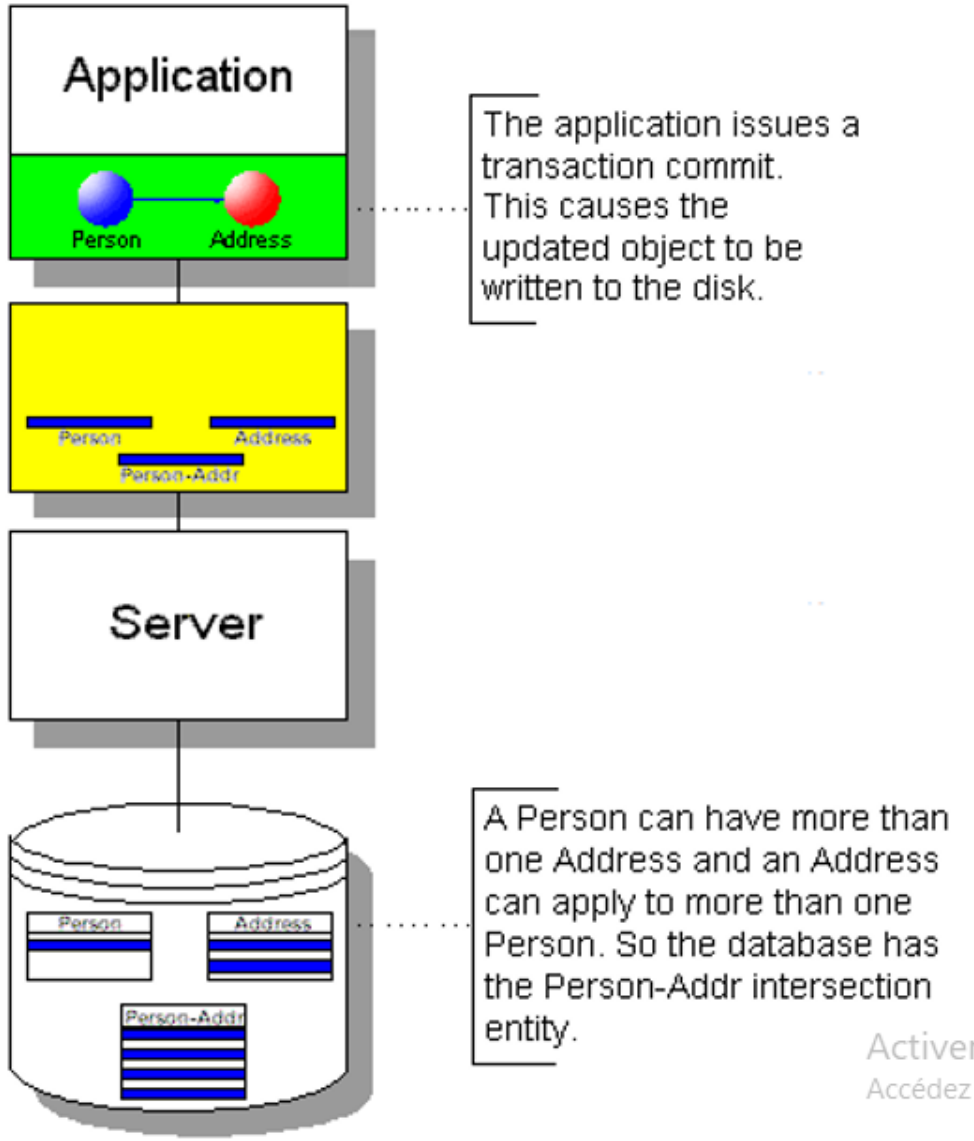
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

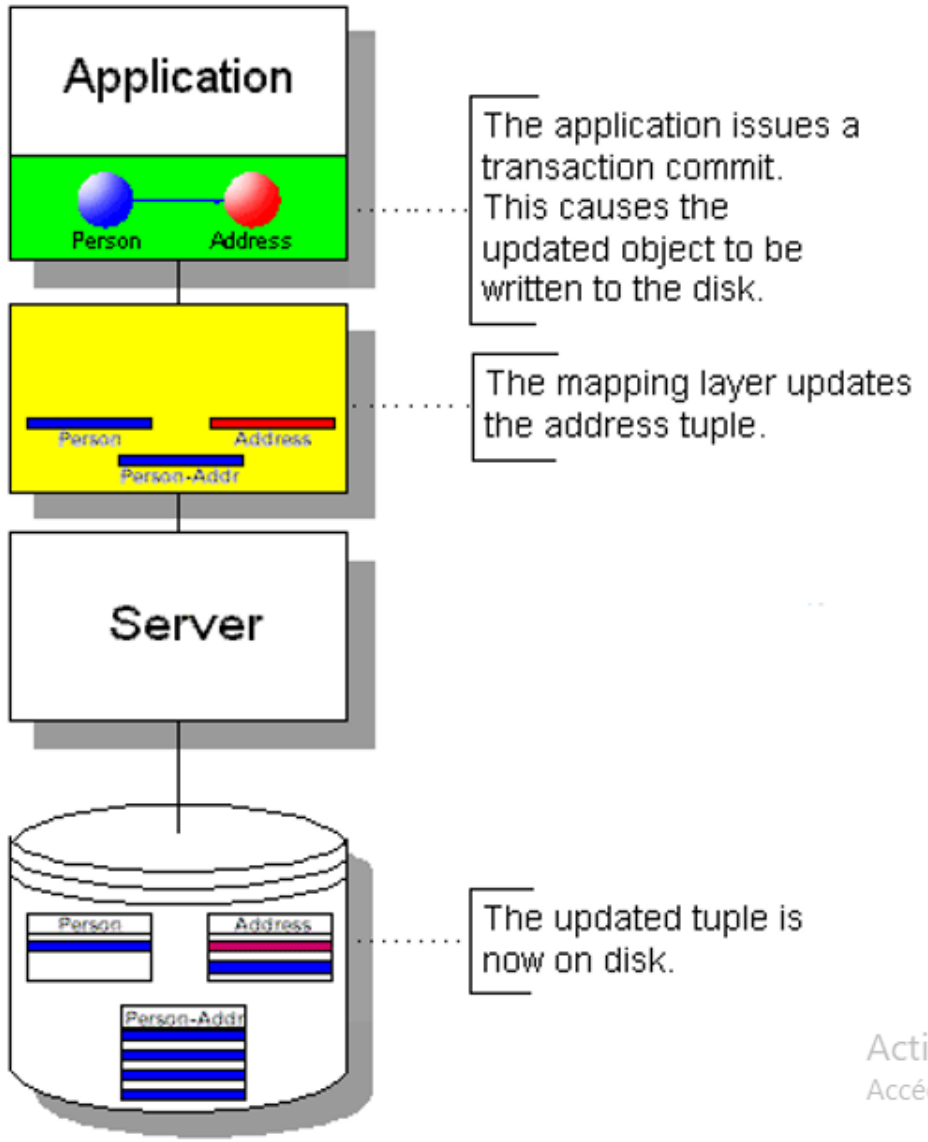
Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Exemple de correspondance Objet/relationnel

Persistance *presque* transparente



Repris de ***http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html***

Non correspondance Objet-Relationnel

- 30% du coût de développement consacré à la mise en correspondance
- Modélisation relationnelle tributaire de la théorie relationnelle
- Modélisation orientée-objet sans définition mathématique rigoureuse ni partie théorique
- Modèles architecturaux ou basés sur les motifs vus comme une solution partielle au problème de non-correspondance : ex. *Entity beans*, *DAO* (*Data Access Object*)
- Réduction du code de correspondance par les outils ORM

Solutions ORM (1/4)

Normes Java :

- **EJB (*Entreprise Java Beans*)** :
  - Gestion de la persistance par conteneur (CMP- *Container-Managed Persistence* et BMP – *Beans Managed Persistence*)
  - Spécifications EJB3.0 (JSR 220 Mai 2006)
- **JPA (*Java Persistence API*)** : Partie des spécifications EJB 3.0 (JSR 220 en Mai 2006) concernant la persistance des composants – JPA 2.2 (Java Persistence API) de Jakarta EE
- **JDO (*Java Data Object*)** :
  - Spécification de Sun 1999 – JDO 2.0 (JSR243 Mars 2006)
  - Abstraction du support de stockage
  - Implémentation libre : **DataNucleus** (ex. JPOX) – dernière version 2020

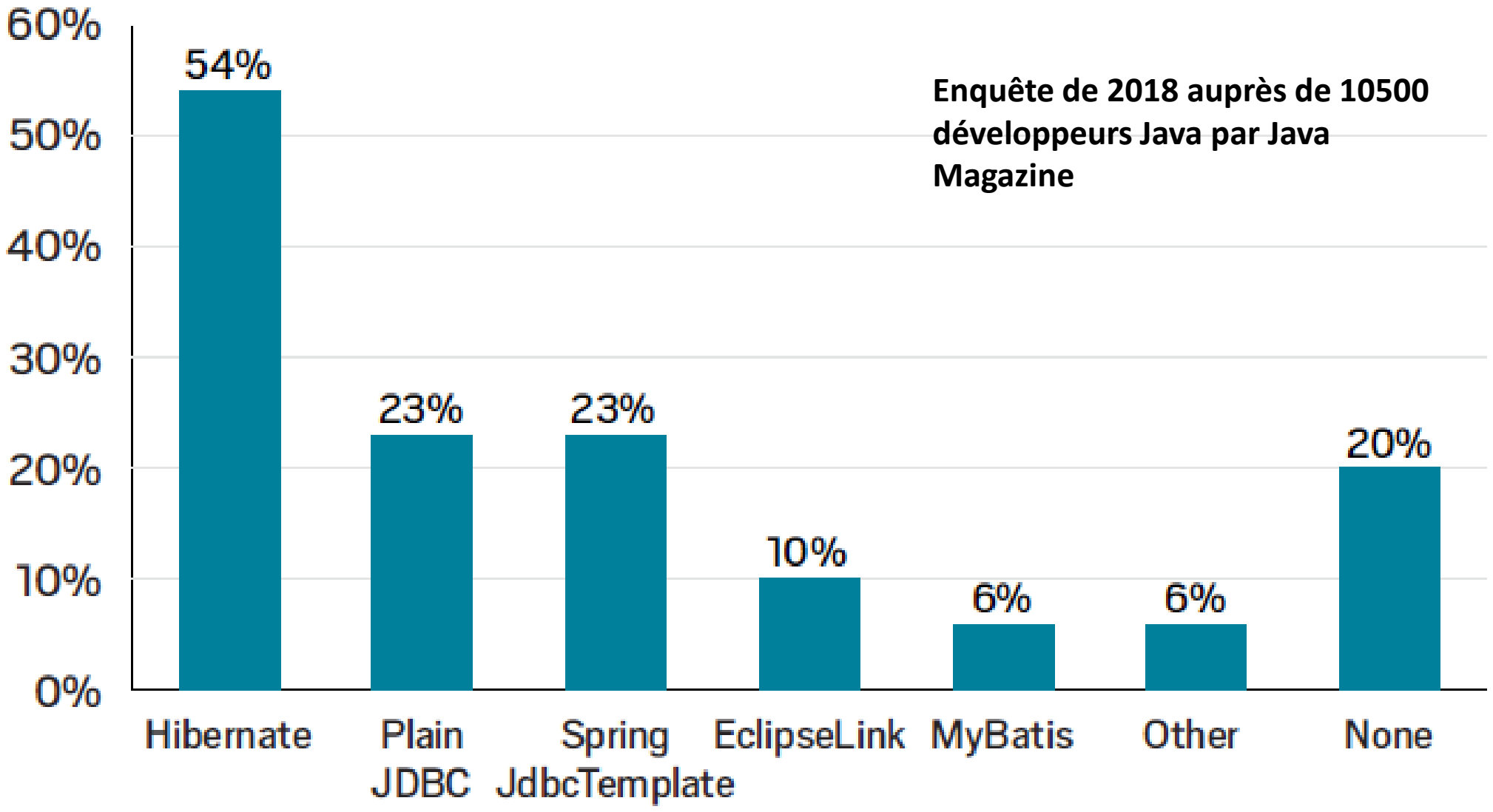
Implémentation de JPA :

- **Hibernate** (***JBoss***) : Solution libre faisant partie du serveur d'appli. JBoss – implémentant les spécifications JSR 220 – complète et bien documentée - plugin Eclipse - Gavin King (fondateur) membre de groupes d'expert d'EJB3
- **TopLink** (***Oracle***) : Solution propriétaire utilisée par la serveur d'application d'Oracle - **TopLink Essentials** *:* version libre disponible dans *Netbeans* ou le serveur d'application (Java EE 5) *Glassfish* de Sun, intégrée dans le projet *EclipseLink*
- **EclipseLink** (issu de Toplink)
- **MyBatis (*Apache*)**

Autres outils :

- **Outils pour d'autres langages :** SQLAlchemy, Dgango, SQLObject pour Python, nHibernate pour .Net …
- **cf. https://en.wikipedia.org/wiki/Comparison_of_object-relational_mapping_software**

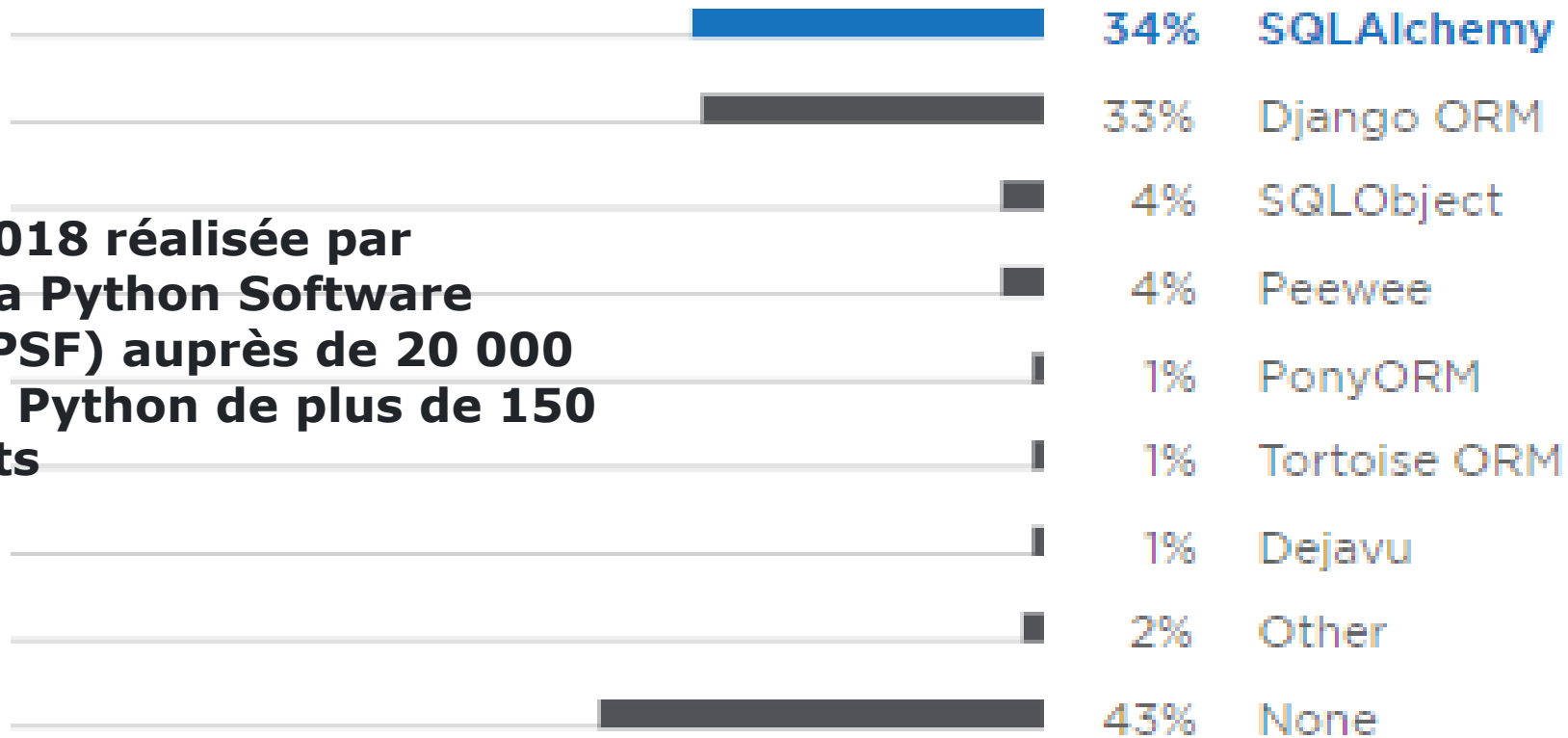
Solutions ORM (2/4)



Enquête de 2018 auprès de 10500 développeurs Java par Java Magazine

Solutions ORM (3/4)

Enquête de 2018 réalisée par JetBrains et la Python Software Foundation (PSF) auprès de 20 000 développeurs Python de plus de 150 pays différents



Solutions ORM (4/4)

Objectifs :

Automatiser et faciliter la correspondance entre les données stockées dans des objets et une base de données relationnelles

- Recherche et enregistrement des données associées à un objet dans une base de données
- Détection de la modification d'un objet et enregistrement des mises à jour en optimisant les accès à la base


Moins de code répétitif à écrire : gain de 30 à 40% du nombre de lignes de code


Amélioration de la portabilité du code en cas de changement de SGBD

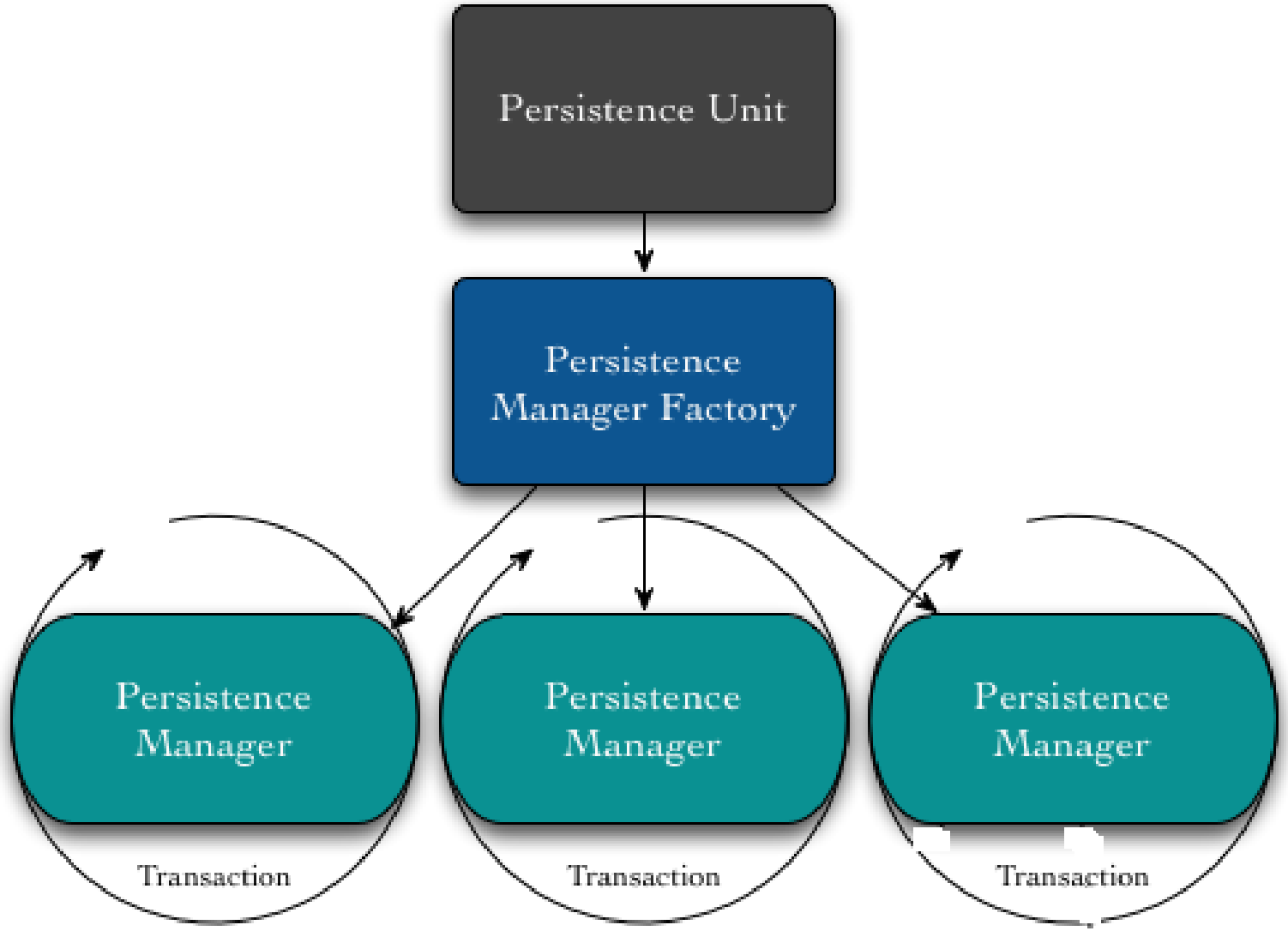

Développement objet (sans penser en terme relationnel)


Possibilité d'avoir un modèle objet fin (pouvant nécessiter un codage à la main complexe pour la persistance)

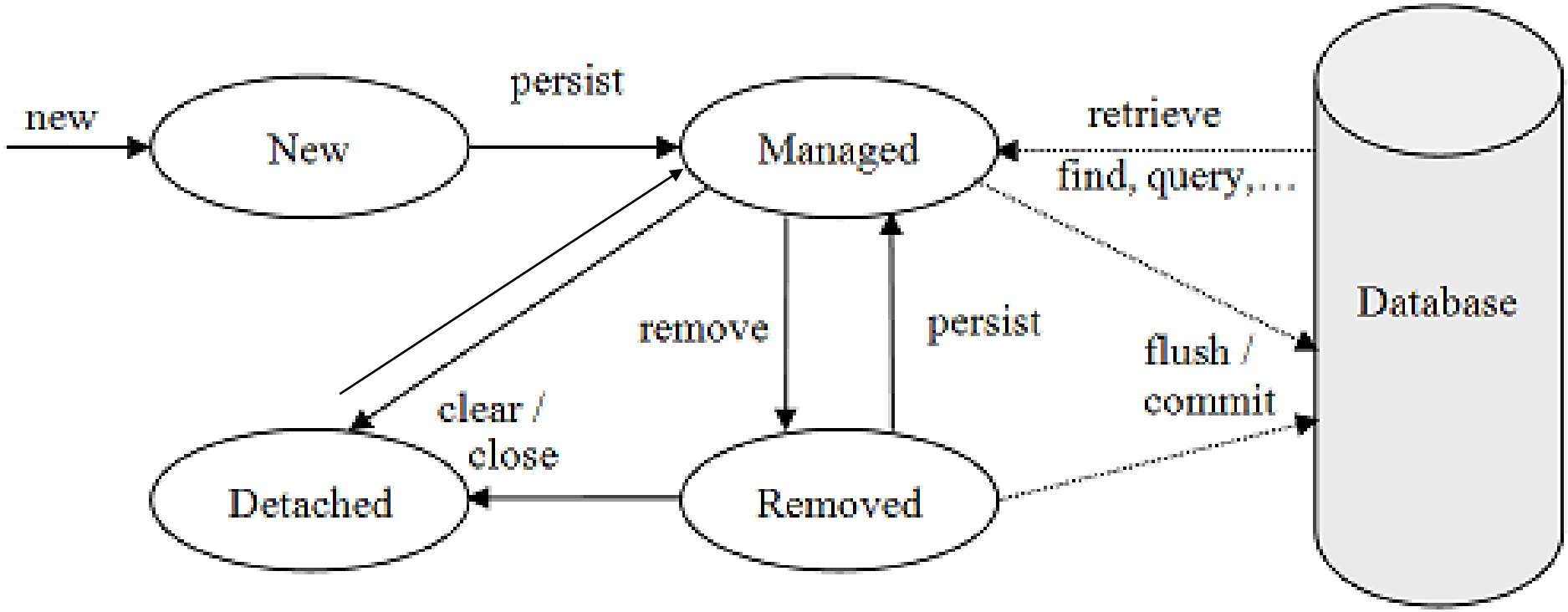

Refactorisation *(Refactoring)* du schéma de la base de données ou du modèle objet facilité


Pas optimal pour des applications modifiant beaucoup de nuplets à chaque update ou comportant essentiellement des requêtes d'agrégation (group by) – Ex. OLAP ou *data mining*

Les 4 piliers de JPA



Cycle de vie des objets (1/2)



États définis par rapport à un *contexte de persistence* (*Entity Manager*)

Cycle de vie des objets (2/2)

- **Temporaire/Nouveau (*New*) :**
  - Instance non associée (et n'ayant jamais été associée) à un contexte de persistance
  - Instance sans identité persistante (i.e. valeur de clé primaire)

- **Persistant (*Managed*):**
  - Instance associée à un contexte de persistance
  - Instance possèdant une identité persistante (i.e. valeur de clé primaire) et, peut-être, un enregistrement/nuplet correspondant dans la base
  - Pour un contexte de persistance particulier, *garantie* de l'équivalence entre l'identité persistante et l'identité Java (i.e. emplacement mémoire de l'objet)

- **Détaché (*Detached*) :**
  - Instance ayant été associée au contexte de persistance à présent fermé ou instance ayant été sérialisée vers un autre processus
  - Instance possédant une identité persistante et peut-être un enregistrement/nuplet correspondant dans la base
  - Aucune garantie concernant la relation entre l'identité persistante et l'identité Java

Exemple simple de correspondance

Implémentation **POJO (*Plain Old Java Object*)** de la classe `Departement` :

```
public class Departement implements java.io.Serializable {
  // Fields
  private int departementId;
  private String nomDepartement;
  /** default constructor */
  public Departement() {}
  /** full constructor */
  public Departement(int departementId, String nomDepartement) {
    this.departementId = departementId;
    this.nomDepartement = nomDepartement;
  }
  // Property accessors
  …
}
```

Relation de bases de données `Departement` :

```
CREATE TABLE Departement
(
  departement_id int4 NOT NULL,
  nom_departement varchar(25) NOT NULL
)
```

Clé primaire ??

departement_id int4	nom_departement varchar
1	MIDO
2	LSO
3	MSO
4	LANGUES

1. Identification des objets (1/5)

Dans le monde objet :

- Identification des objets par l'adresse de leur emplacement mémoire
- 2 notions différentes :
  - Identité d'objets : a==b (notion définie par la machine virtuelle Java)
  - "Égalité" ou équivalence : méthode *equals()* à implémenter dans la classe

Dans le modèle relationnel :

- Identification des nuplets par la valeur de la clé primaire
- Pas de possibilité d'avoir deux nuplets avec les mêmes valeurs dans une relation

1. Identification des objets (2/5)

En cas d'utilisation d'un SGBD relationnel-objet (ex. PostgreSQL) :

- Ajout automatique d'un attribut OID pour distinguer les nuplets (utilisé comme clé primaire dans les relations système)
- Possibilité de le choisir comme clé primaire dans une relation utilisateur

```
CREATE TABLE departement
(
  departement_id int4 NOT NULL,
  nom_departement varchar(25) NOT NULL,
  CONSTRAINT pk_departement PRIMARY KEY (oid)
)
WITH OIDS;
```

oid	departement_id int4	nom_departement varchar
18494	1	MIDO
18495	2	LSO
18496	3	MSO
18497	4	LANGUES

1. Identification des objets (3/5)

En cas d'utilisation d'un SGBD relationnel pur :

Préférer les clés primaires sans contrepartie dans le monde réel ou clé de substitution (ou artificielle - *surrogate key*)

- Pour éviter les clés composées de plusieurs attributs
- Pour faciliter l'indexation (ex. un entier incrémenté automatiquement)
- Pour faciliter les mises à jour (ne pas avoir à changer toutes les clés étrangères y faisant référence)
- Pour éliminer le lien avec le modèle métier

Il faut assurer la cohérence des données en ajoutant une contrainte d'unicité pour les attributs identificateurs métier

1. Identification des objets (4/5)

Objet persistant = représentation en mémoire d'un nuplet

Un même nuplet ne doit pas être représenté par plusieurs objets en mémoire centrale pour une même session de travail

Exemple :
Création en mémoire d'un objet e1 de la classe `Enseignant` *(à l'occasion d'une navigation à partir d'un objet* `Enseignement`*)*
Possibilité de retrouver le même enseignant depuis un autre `Enseignement` *ou depuis un* `Département`

⇒ *Ne pas créer d'objet e2 en mémoire centrale indépendant de e1*

⇒ Utilisation du **cache**

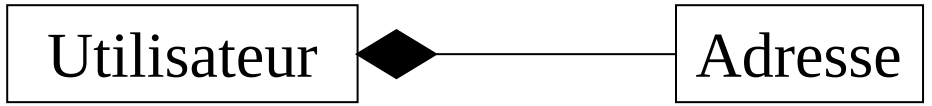
- "Index" des objets créés en mémoire
  (avec conservation de l'identité relationnelle – clé primaire)
- Recherche dans le cache avant toute récupération dans la base

1. Identification des objets (5/5)

Différence de granularité entre les deux modèles

- Modèle objet de granularité plus fine :
  - " + de classes que de relations "
  - Instances de plusieurs classes (dépendantes) sauvegardés dans la même relation

Ex.



⇒ *Pas d'obligation de créer une relation Adresse dans la base*

Si insertion des attributs de Adresse dans la relation Utilisateur ⇒ des objets Adresse sans identification liée à la base

- Possibilité d'utiliser les *User Defined Type* de SQL99 – mais problème de compatibilité et de standardisation dans les SGBD

2. Traduction des associations (1/5)

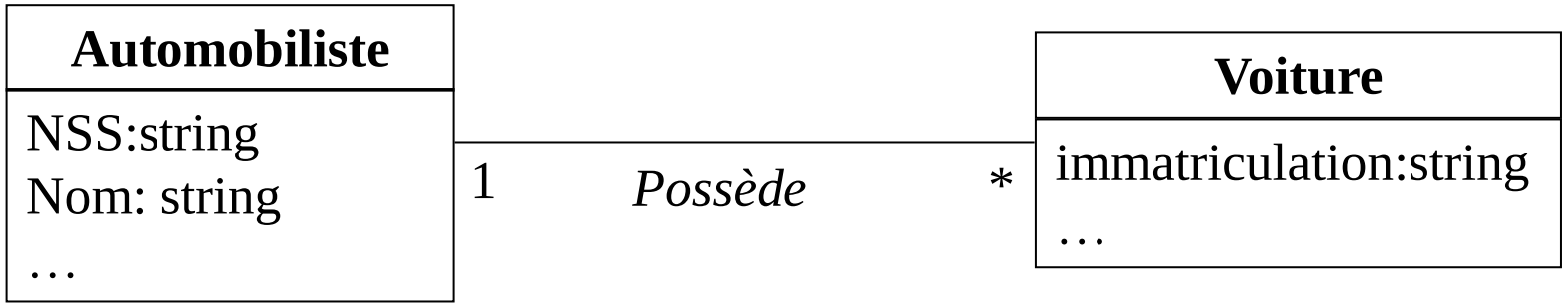
Dans le monde objet :

- Association = ensemble de liens entre objets
- 3 types de représentation :
  - variable d'instance de type objet (multiplicité 1 ou *..1)
  - variable d'instance de type collection d'objets (multiplicité 1..* ou *)
  - Classe-association (multiplicité *)
- Références d'objet par nature unidirectionnelle
- Pas de déduction de la multiplicité à partir de la classe Java (si codage unidirectionnel)

Dans le modèle relationnel

- Représentation des associations par :
  - Une ou plusieurs clés étrangères (multiplicité 1 ou *..1 ou 1..*)
  - Une relation dont la clé primaire est composée de clés étrangères (multiplicité *) : table de liaison ou table d'association
- Bi-direction par jointure entre les relations
- Déduction de la multiplicité par analyse de la définition des clés étrangères

2. Traduction des associations (2/5)



En cas d'implémentation bidirectionnelle ou d'un seul côté pour une implémentation uni-directionnelle

De Automobiliste vers Voiture

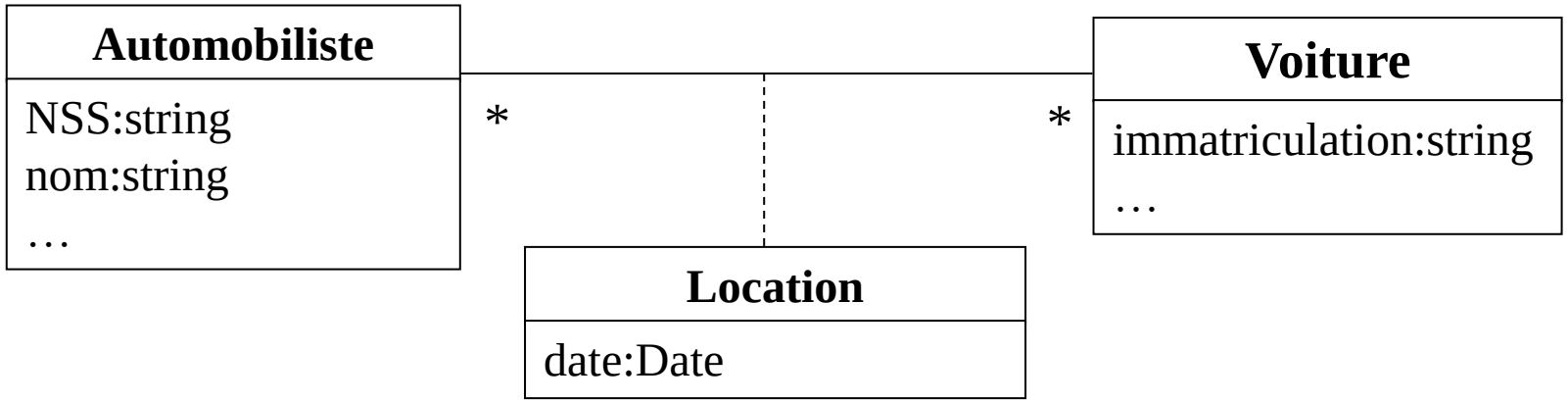
De Voiture vers Automobiliste

```
public class Automobiliste {
  // Fields
  private String NSS;
  private String nom;
  private Collection<Voiture> parc_automobile;
  …
}
public class Voiture {
  // Fields
  private String immatriculation;
  private Automobiliste proprietaire;
  …
}
```

2. Traduction des associations (3/5)


```
CREATE TABLE Automobilsite
( Automobiliste_ID  SERIAL,
  NSS varchar(10) NOT NULL,
  Nom varchar(20) NOT NULL,
  …
  CONSTRAINT PK_Automobiliste PRIMARY KEY (Automobiliste_ID),
) ;

CREATE TABLE Voiture
( Voiture_ID  SERIAL,
  Immatriculation varchar(10) NOT NULL,
  …
  Proprietaire_ID int NOT NULL,
  CONSTRAINT PK_ Voiture PRIMARY KEY (Voiture_ID),
  CONSTRAINT FK_Automobiliste_Voiture
   FOREIGN KEY (Proprietaire_ID) REFERENCES Automobiliste (Automobiliste_ID)
);
```

2. Traduction des associations (4/5)




```
public class Location {
  private Automobiliste loueur;
  private Voiture vehicule;
  private Date date;
  …
}
```

```
CREATE TABLE Location
( …
 CONSTRAINT PK_Location PRIMARY KEY (Automobiliste_ID,Voiture_ID, Date),
 CONSTRAINT FK_Location_Voiture
   FOREIGN KEY (Voiture_ID) REFERENCES Voiture (Voiture_ID),
 CONSTRAINT FK_Location_Automobiliste
   FOREIGN KEY (Automobiliste_ID) REFERENCES Automobiliste (Automobiliste_ID),
);
```

2. Traduction des associations (5/5)

- Gestion des associations plus complexe en objet qu'en relationnel

  Ex. Modification d'un propriétaire de voiture

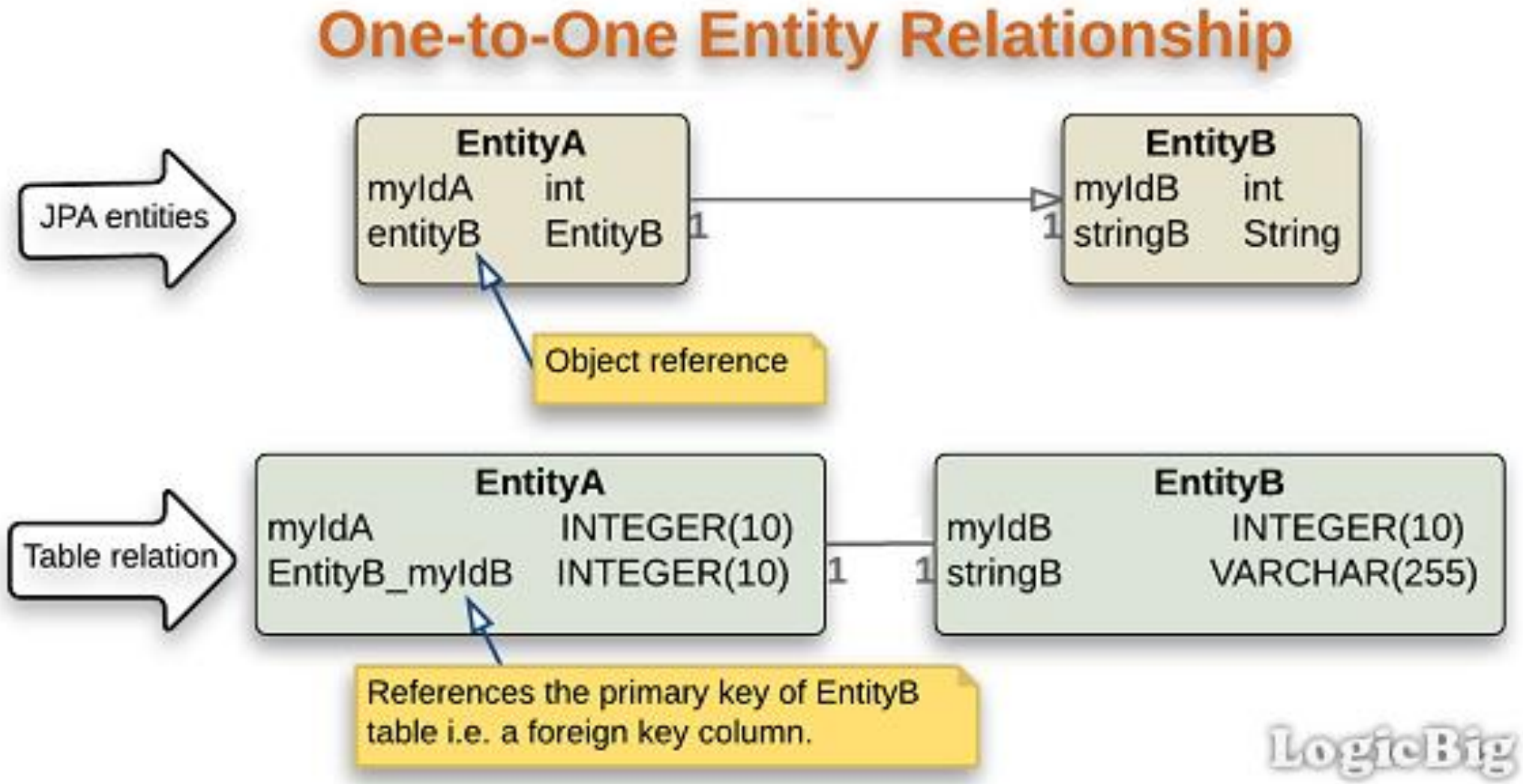
  ⇒*Modification de la valeur de la clé étrangère dans la relation Voiture*

  ⇒*Suppression du véhicule correspondant dans la collection parc_automobile de l'objet Java Automobiliste*

  + mise à jour de l'instance propriétaire de l'objet Voiture

- Gestion automatique dans EJB 2.x
- Choix de ne rien automatiser dans JPA

Annotation JPA (association) : *1-1* unidirectionnelle (java vs relationnel avec clé étrangère)

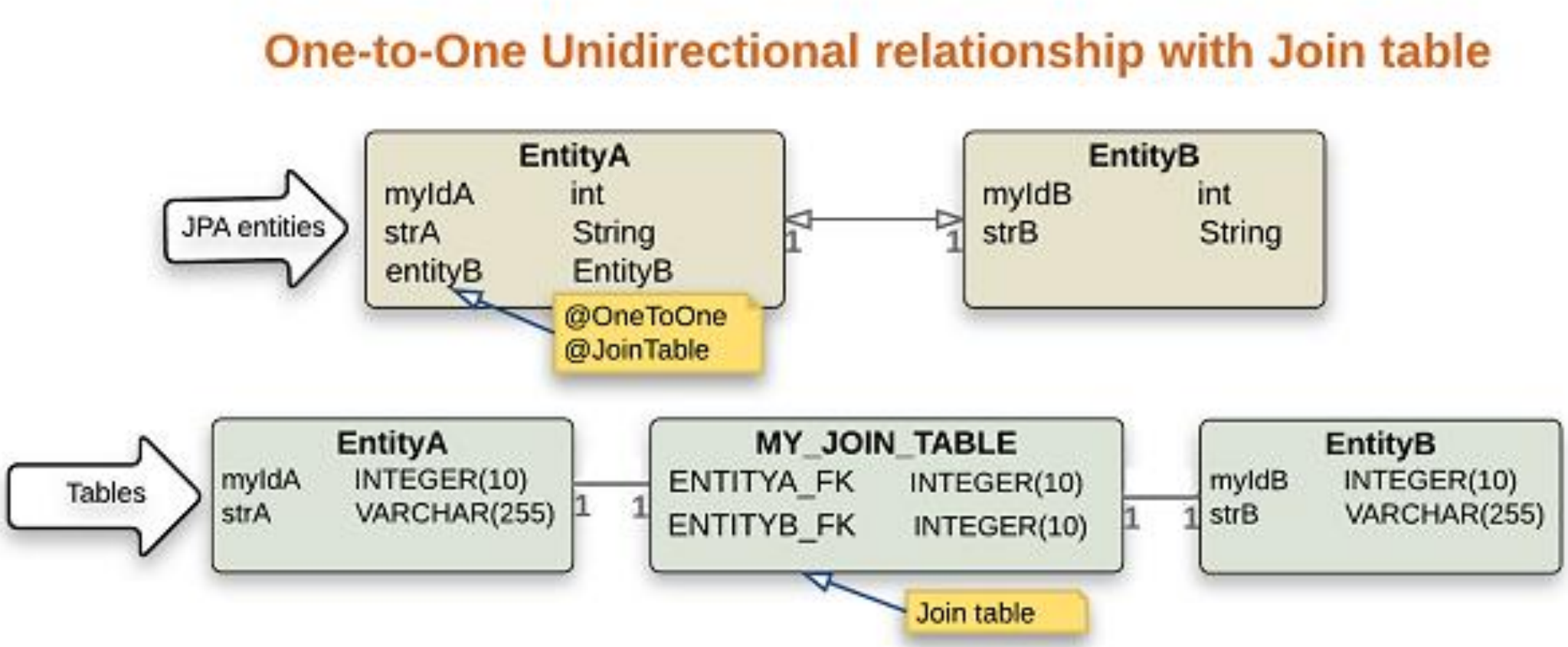


Annotation JPA (association) : *1-1* unidirectionnelle traduite en clé étrangère (en java)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  @OneToOne
  private EntityB entityB;

  public EntityB getEntityB() {
      return entityB;
  }

  public void setEntityB(EntityB entityB) {
      this.entityB = entityB;
  }
}
```

```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  private String stringB;

  public String getStringB() {
      return stringB;
  }

  public void setStringB(String stringB) {
      this.stringB = stringB;
  }
}
```

Annotation JPA (association) : *1-1* unidirectionnelle (java vs relationnel avec table-association)

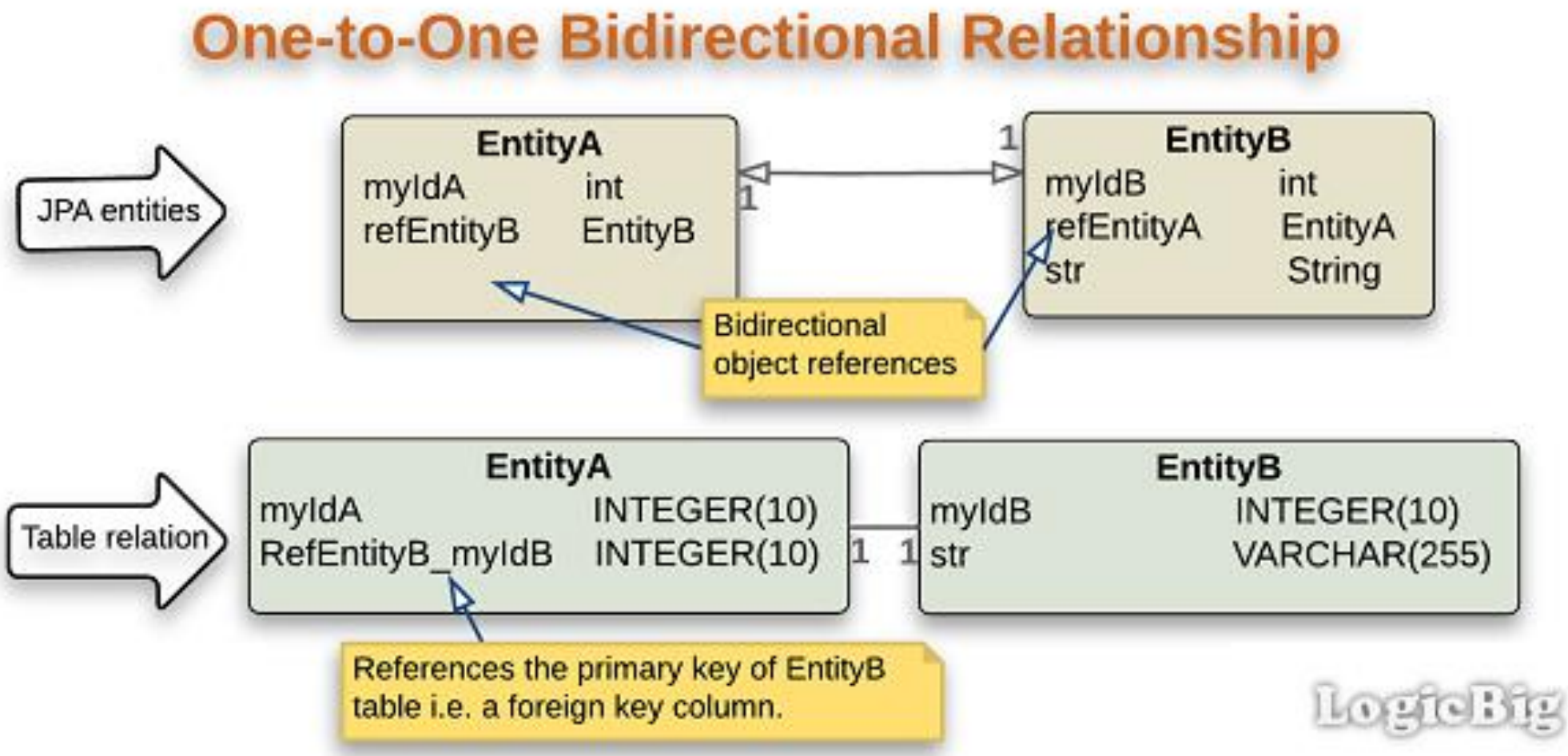


Annotation JPA (association) : *1-1* unidirectionnelle traduite en table association (en java)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  private String stringA;
  @OneToOne
  @JoinTable(name = "MY_JOIN_TABLE",
          joinColumns = {
                  @JoinColumn(name = "ENTITYA_FK", referencedColumnName = "myIdA")
          },
          inverseJoinColumns = {
                  @JoinColumn(name = "ENTITYB_FK", referencedColumnName = "myIdB", unique = true)
          }
  )
  private EntityB entityB;
    .............
}
```

```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  private String stringB;
    .............
}
```

Annotation JPA (association) : *1-1* bidirectionnelle (java vs relationnel avec clé étrangère)

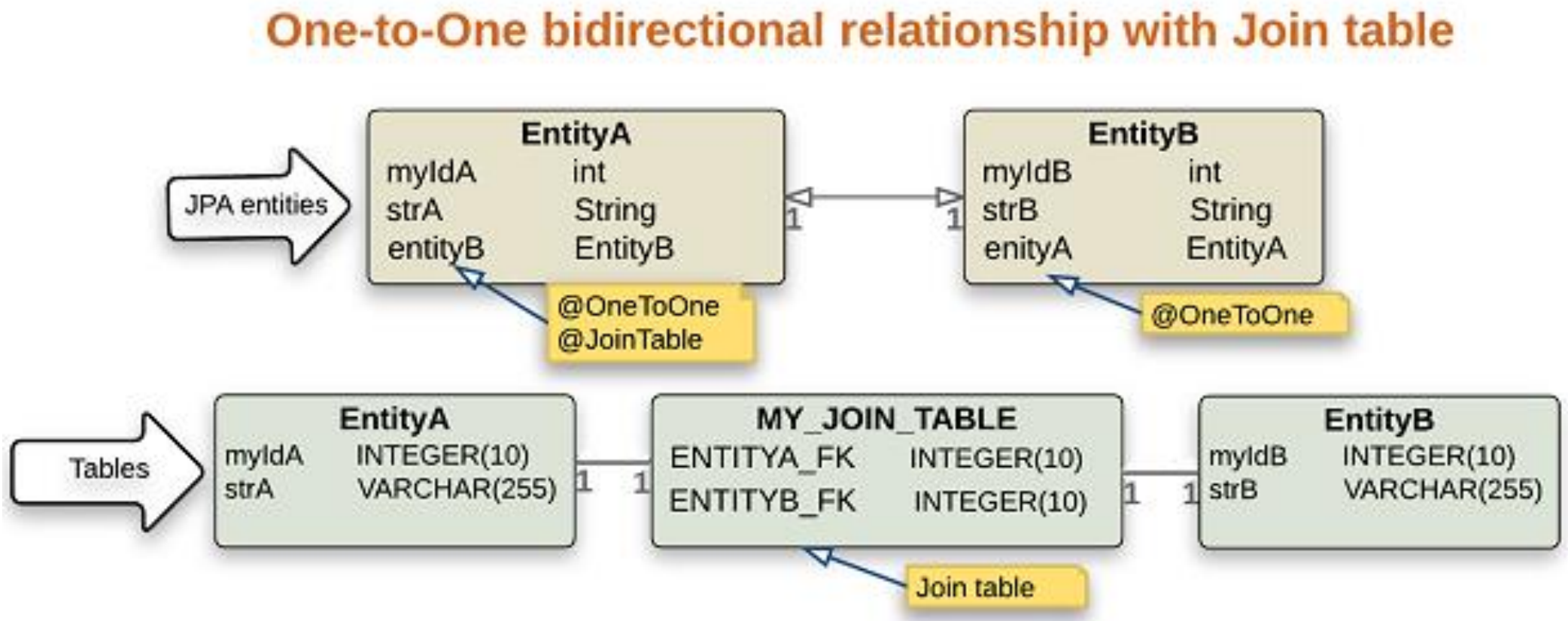


Annotation JPA (association) : *1-1* unidirectionnelle traduite en clé étrangère (en java)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  @OneToOne
  private EntityB refEntityB;
    ..............
  @Override
  public String toString() {
      return "EntityA{" +
              "myIdA=" + myIdA +
              '}';
  }
  //getters/setters
}
```

```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  @OneToOne(mappedBy = "refEntityB")
  private EntityA refEntityA;
  private String str;
    ..............
  @Override
  public String toString() {
      return "EntityB{" +
              "myIdB=" + myIdB +
              ", str='" + str + '\'' +
              '}';
  }
  //getters/setters
}
```

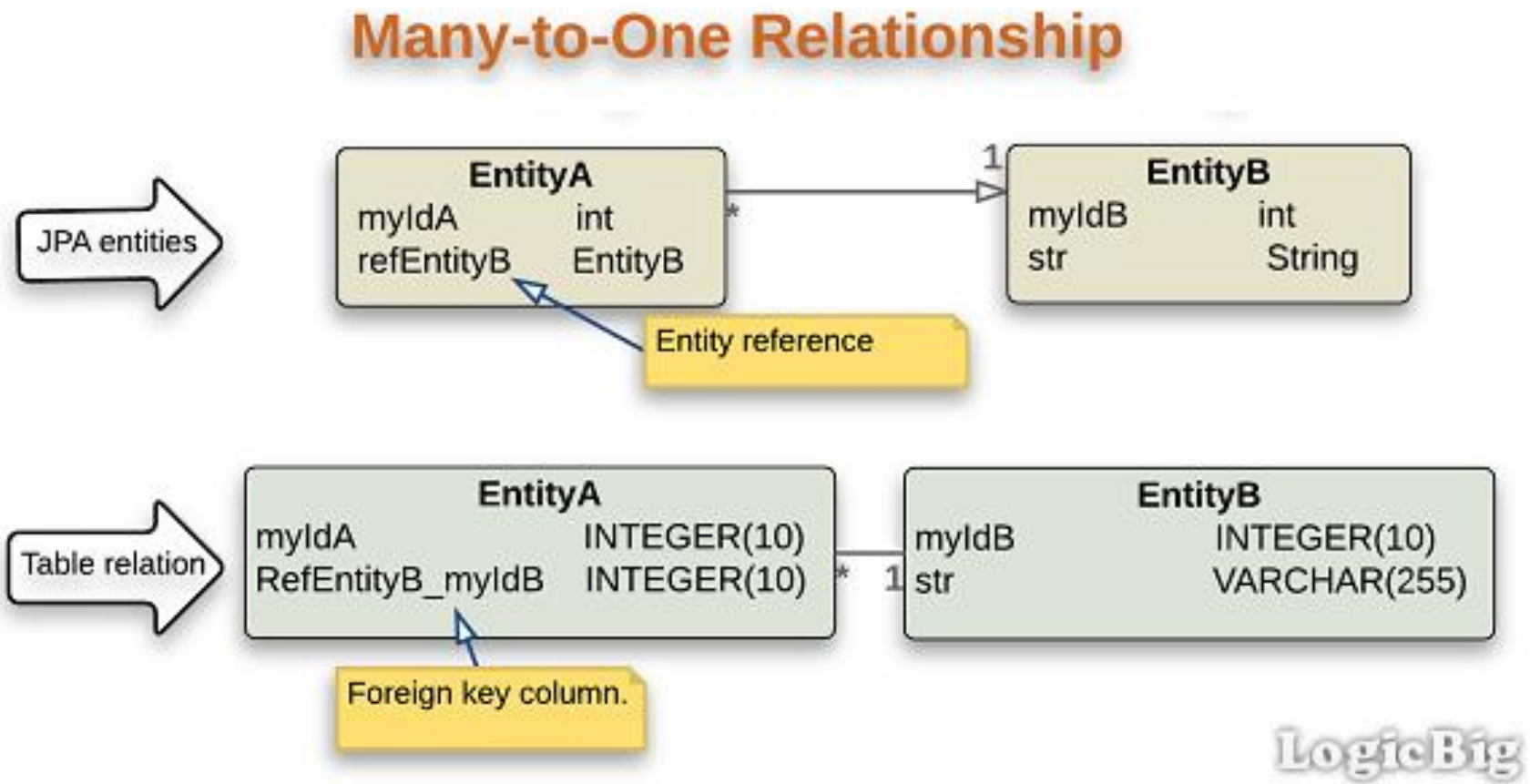
Annotation JPA (association) : *1-1* bidirectionnelle (java vs relationnel avec table-association)



Annotation JPA (association) : *-*1* unidirectionnelle (java vs relationnel avec clé étrangère)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  @ManyToOne
  private EntityB refEntityB;
    ............
}
```

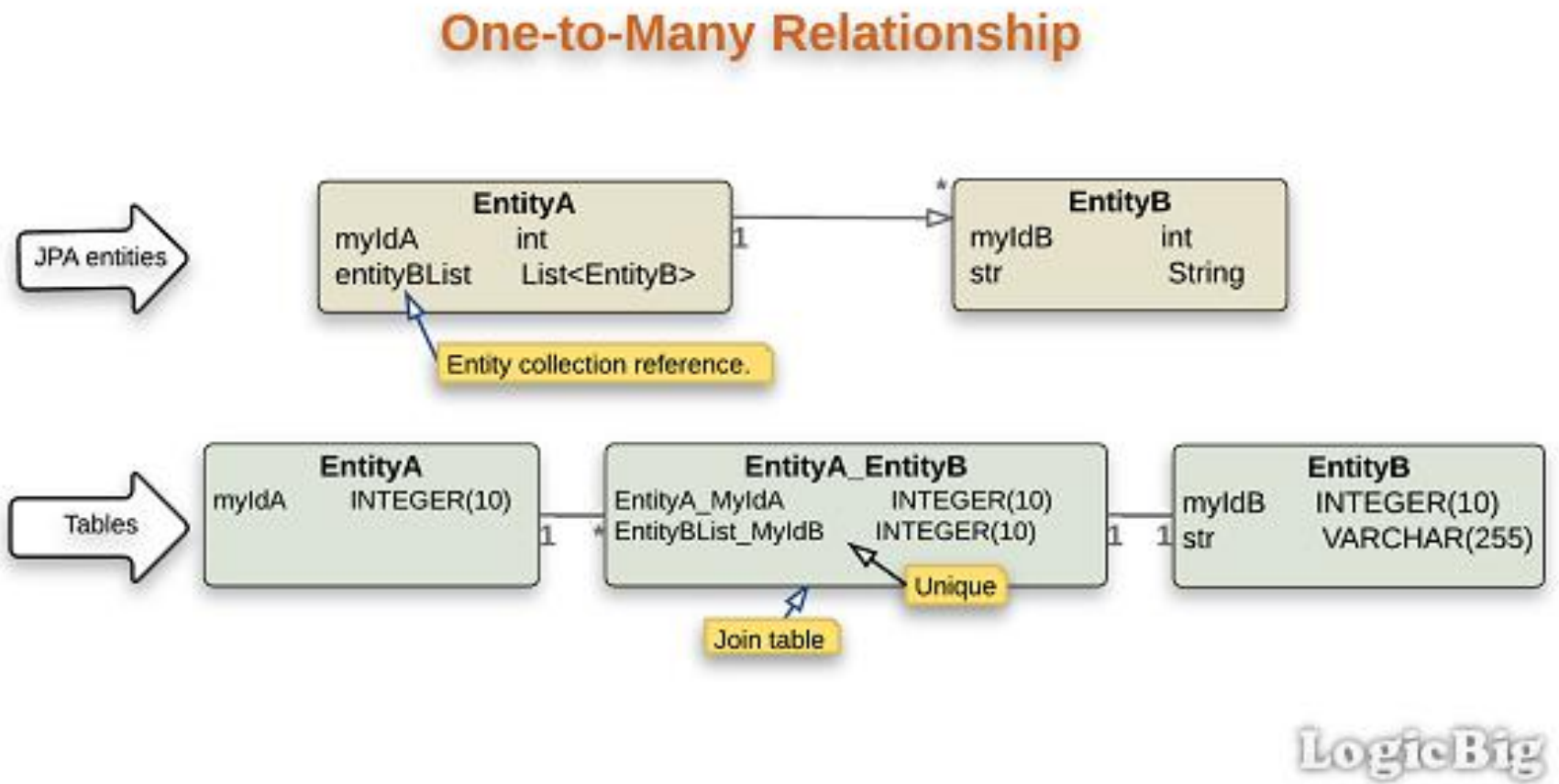
```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  private String str;
    ............
}
```



Annotation JPA (association) : *1*-* unidirectionnelle (java vs relationnel avec table association)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  @OneToMany
  private List<EntityB> entityBList;
    ..............
}
```

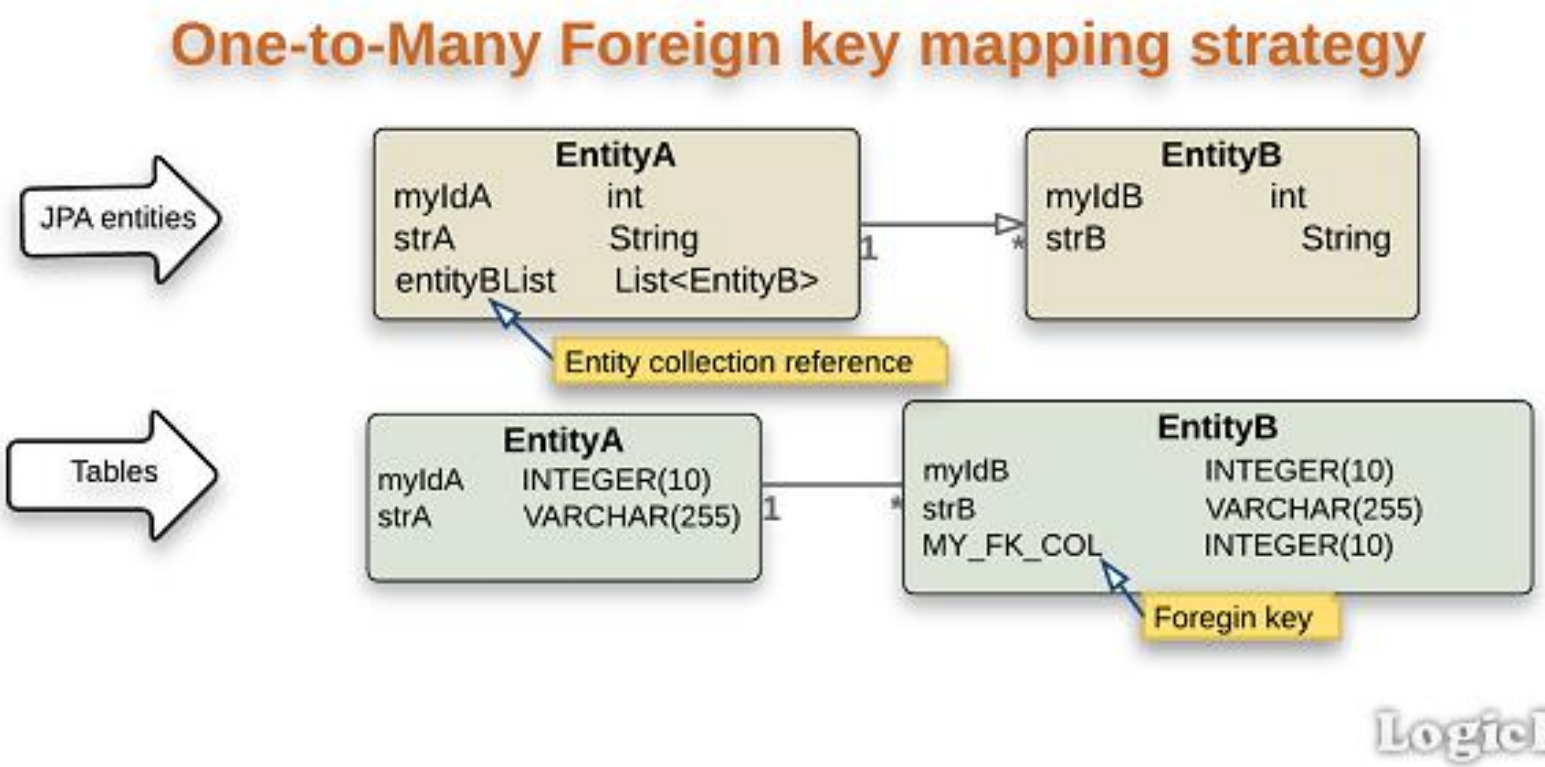
```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  private String str;
    ..............
}
```



Annotation JPA (association) : *1*-* unidirectionnelle (java vs relationnel avec clé étrangère)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  private String strA;
  @OneToMany
  @JoinColumn(name = "MY_FK_COL")
  private List<EntityB> entityBLis
    ..............
}
```

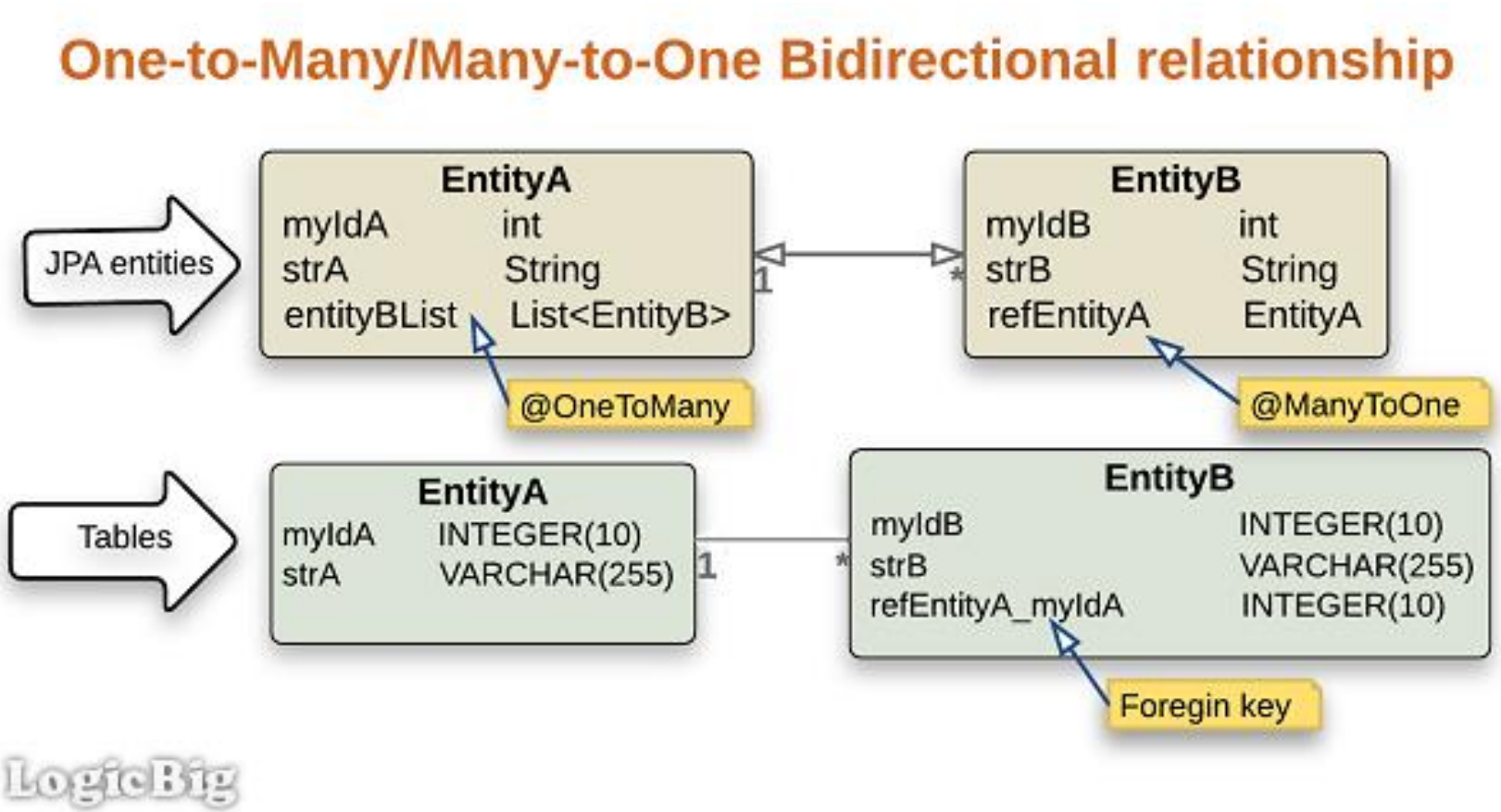
```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  private String strB;
    ..............
}
```



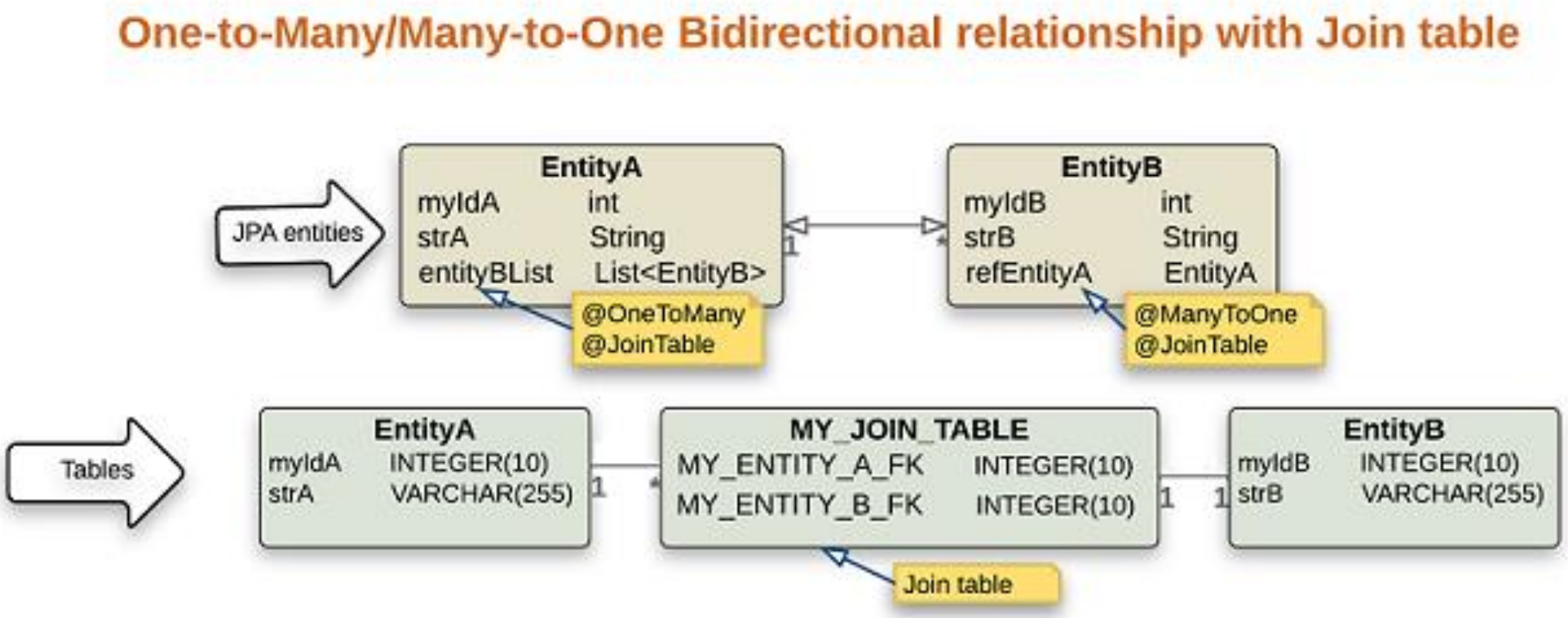
Annotation JPA (association) : **-1 et 1-** bidirectionnelle (java vs relationnel avec clé étrangère)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  private String strA;
  @OneToMany(mappedBy = "refEntityA")
  private List<EntityB> entityBList;
    .............
}
```

```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  private String strB;
  @ManyToOne
  private EntityA refEntityA;
    .............
}
```



Annotation JPA : *-1 et 1-* bidirectionnelle (java vs relationnel avec table association)



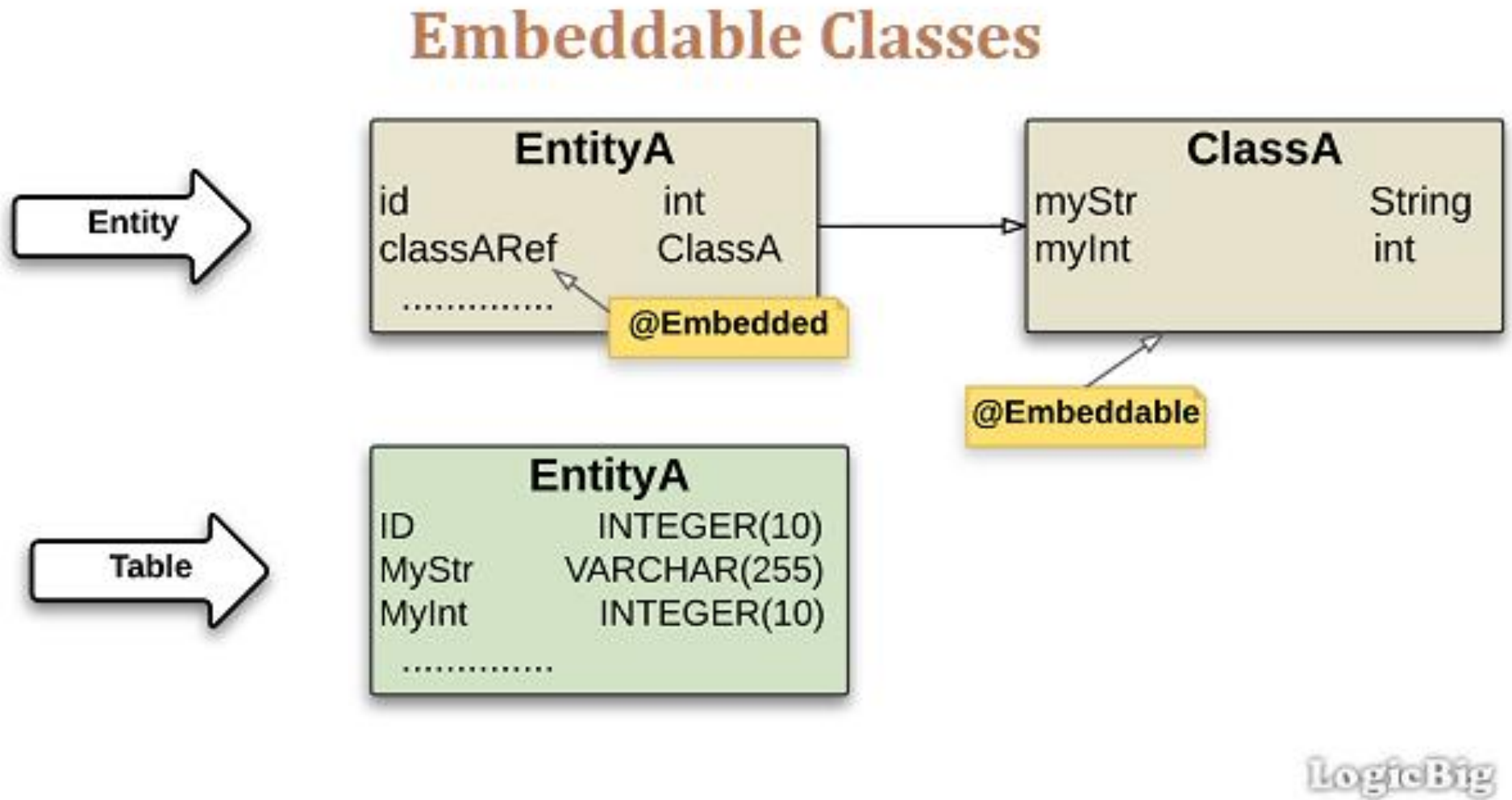
Annotation JPA (association) : Annotation JPA : *1-** bidirectionnelle (en java avec table association)

```java
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int myIdA;
  private String strA;
  @OneToMany
  @JoinTable(name = "MY_JOIN_TABLE",
          joinColumns = {@JoinColumn(name = "MY_ENTITY_A_FK", referencedColumnName = "myIdA")},
          inverseJoinColumns = {@JoinColumn(name = "MY_ENTITY_B_FK", referencedColumnName = "myIdB")}
  )
  private List<EntityB> entityBList;
    ..............
}
```

Annotation JPA (association) : Annotation JPA : *-*1* bidirectionnelle (en java avec table association)

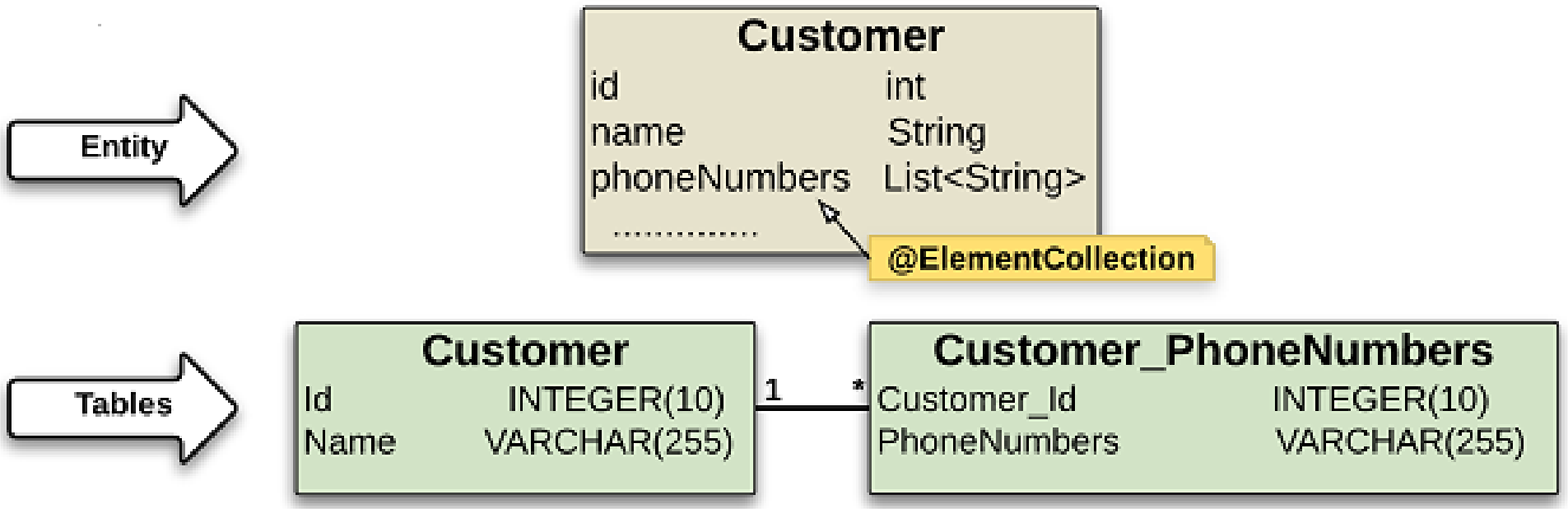
```java
@Entity
public class EntityB {
  @Id
  @GeneratedValue
  private int myIdB;
  private String strB;
  @ManyToOne
  @JoinTable(name = "MY_JOIN_TABLE",
          joinColumns = {@JoinColumn(name = "MY_ENTITY_B_FK", insertable = false,
                  updatable = false, referencedColumnName = "myIdB")},
          inverseJoinColumns = {@JoinColumn(name = "MY_ENTITY_A_FK", insertable = false,
                  updatable = false, referencedColumnName = "myIdA")}
  )
  private EntityA refEntityA;
    ..............
}
```

Annotation JPA (différence de granularité) : classe embarquée



Annotation JPA (différence de granularité) : correspondance de collection de valeurs

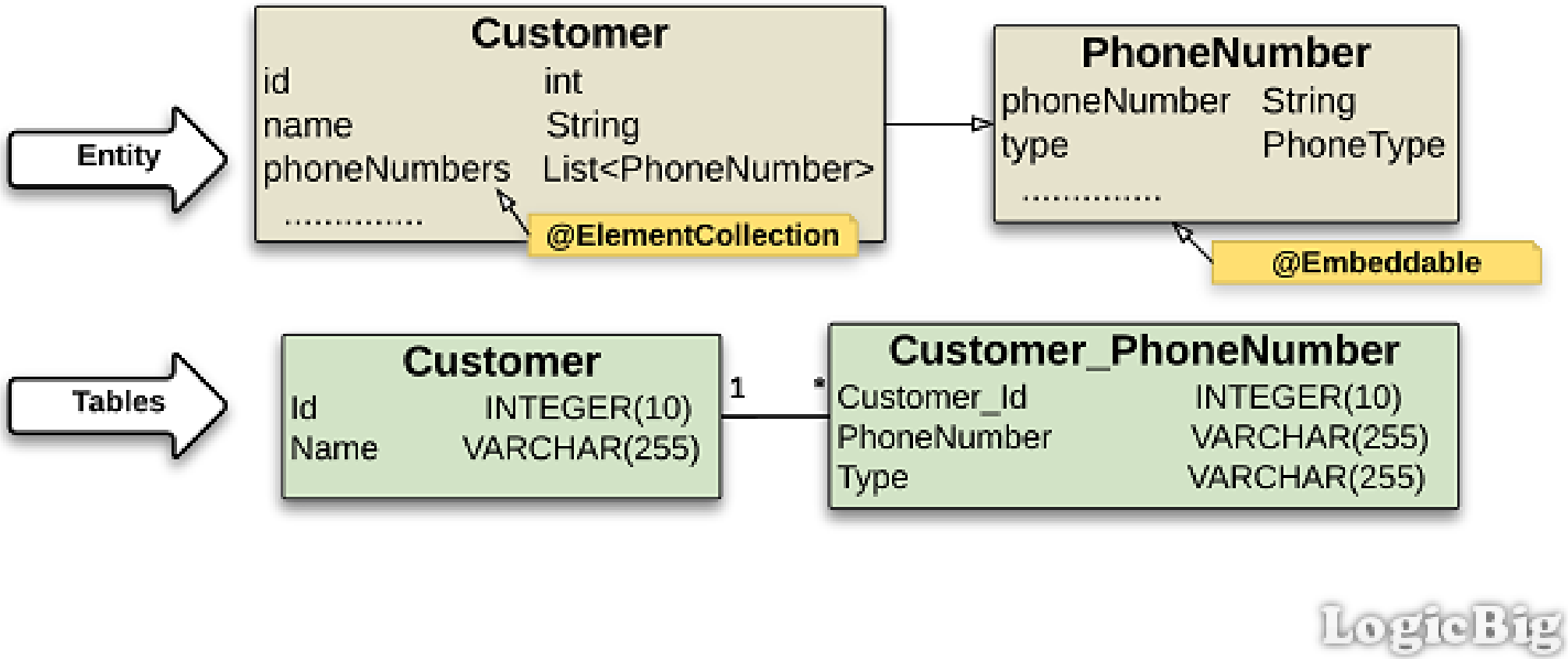
Basic Type Collection Mapping



LogicBig

Annotation JPA (différence de granularité) : correspondance de collection d'objets

Embeddable Collection Mapping

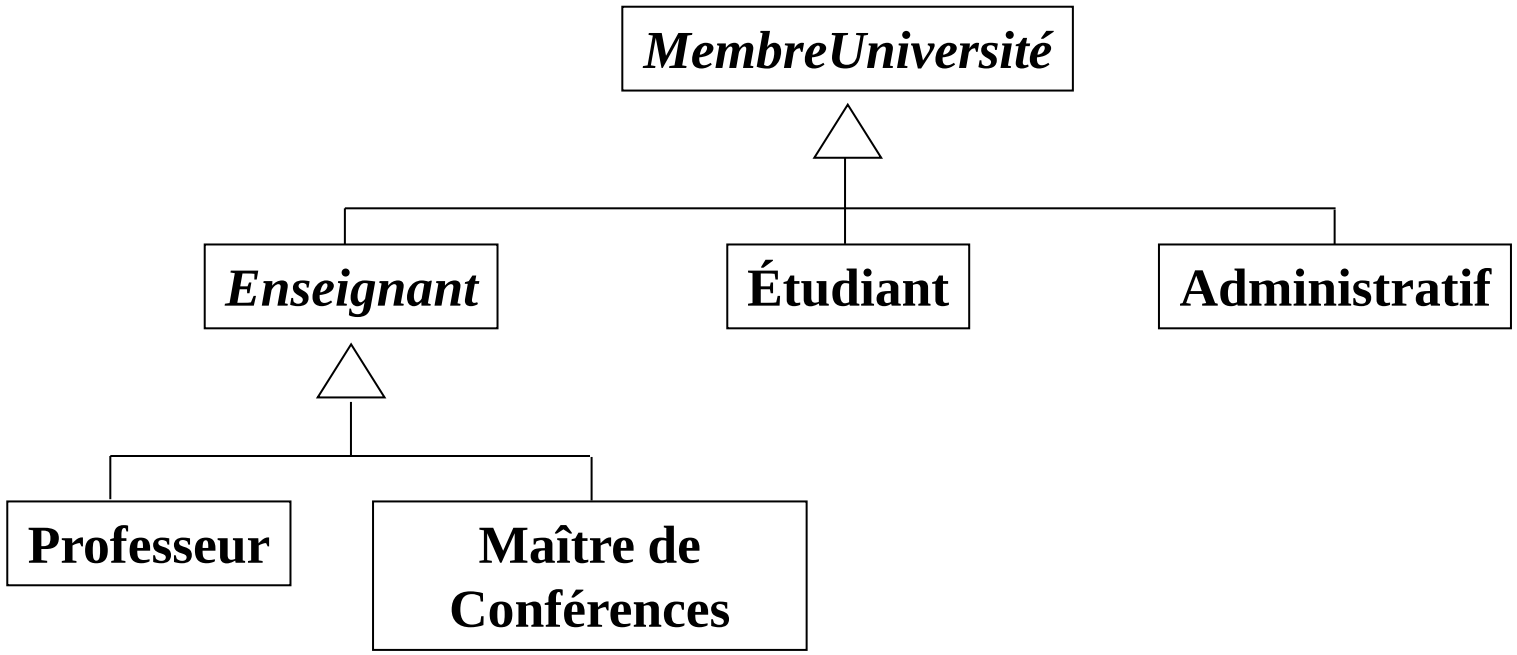


LogicBig

Traduction de l'héritage (1/5) : 3 stratégies

Plusieurs méthodes :

1. Faire correspondre toutes les classes de la hiérarchie à une seule relation de bases de données
2. Représenter chaque classe (abstraite ou concrète) par une relation
3. Représenter chaque classe concrète par une relation



Traduction de l'héritage (2/5) : une seule relation pour toute la hiérarchie

1. Correspondance de toutes les classes de la hiérarchie avec une seule relation de bases de données :

   Ex. une seule relation (correspondant à la racine du graphe) avec les attributs de chaque classe + un attribut type


Facile à mettre en place

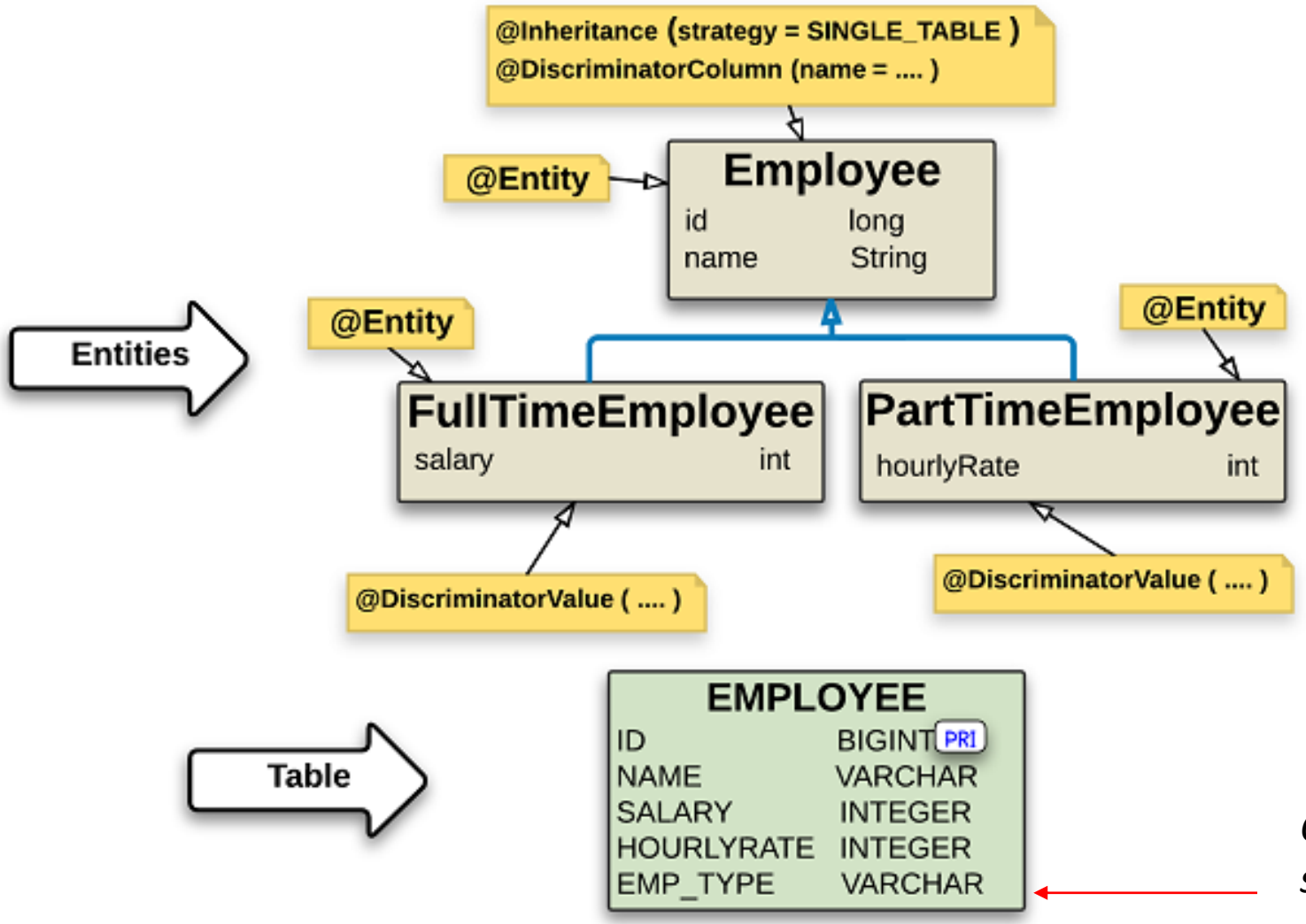

Possibilité de requêtes et associations polymorphes


Obligation de gérer des valeurs NULL pour plusieurs colonnes


Pas de possibilité de déclarer une contrainte NOT NULL sur une de ces colonnes même si la contrainte doit être vérifiée

Annotation JPA : une seule relation/hiérarchie

Single Table Inheritance Strategy



Colonne pour stocker la valeur du discriminateur

Traduction de l'héritage (3/5) : une relation par classe (abstraite ou concrète)

2. Représentation de chaque classe (abstraite ou concrète) par une relation :

⇒ Répartition des attributs d'un objet dans plusieurs relations

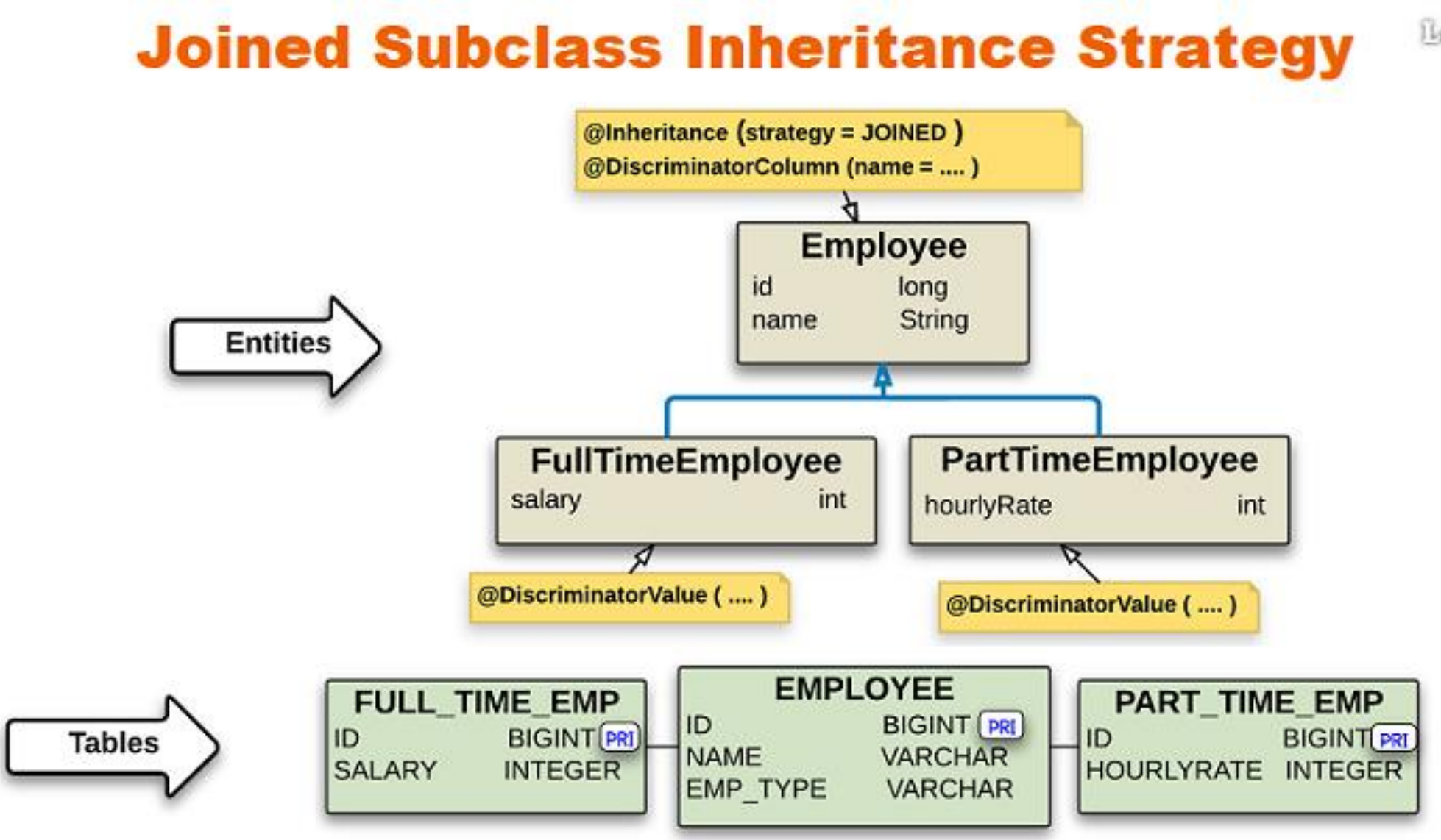
Ex. 1 objet Professeur correspond à un nuplet dans la relation Enseignant et à un nuplet dans la relation Professeur

⇒ Préservation de l'identité en donnant la même valeur de clé primaire à chaque nuplet correspondant à l'objet dans les différentes relations

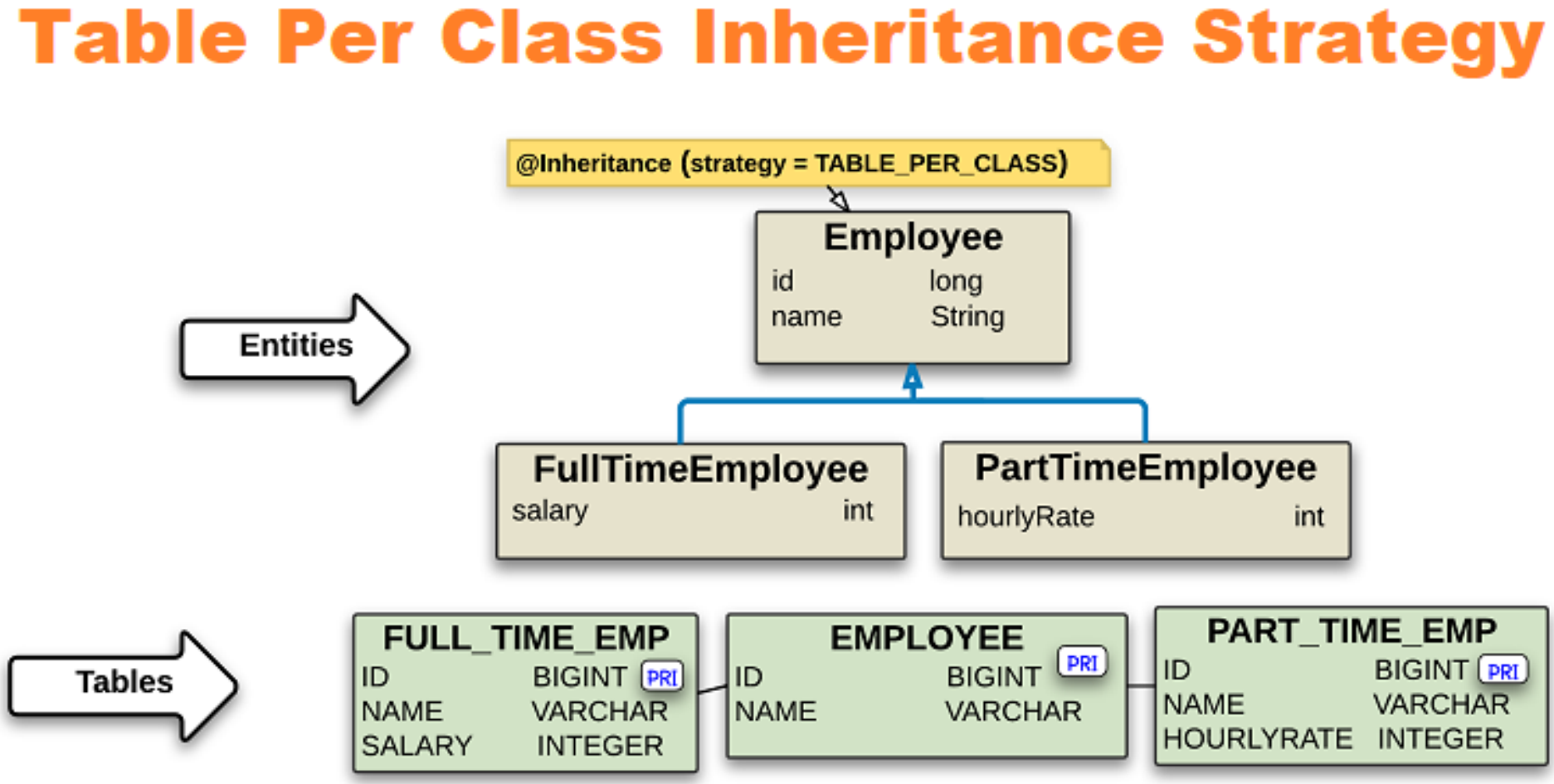
⇒ Clés primaires des relations correspondantes aux classes filles = clés étrangères faisant référence à la clé primaire de la relation correspondant à la classe mère

Ex. la clé primaire de la relation Professeur et aussi une clé étrangère qui fait référence à la clé primaire de la relation Enseignant

Annotation JPA pour une relation/classe : bijection relationnel/hiérarchie avec discriminateur



Annotation JPA pour une relation/classe : bijection relationnel/hiérarchie sans discriminateur



Traduction de l'héritage (4/5) : avantages et inconvénients d'une relation / classe (concrète ou abstraite)

2. Représentation de chaque classe (abstraite ou concrète) par une relation (suite) :


Simple bijection entre les classes et les relations


Possibilité de faire des requêtes et associations polymorphes


Nombreuses jointures à faire en cas de hiérarchie complexe

Possibilité de limiter certains problèmes en ajoutant des attributs dans les classes mères :

Ex. un attribut type dans la relation `MembreUniversité` *pour éviter une jointure lors de la requête « quels sont les noms et prénoms des maîtres de conférences ? »*


Problème de performances


Vérification des contraintes d'intégrité

Traduction de l'héritage (5/5) : une relation par classe concrète uniquement

3. Représentation de chaque classe concrète par une relation :
   - Correspondance de chaque classe concrète avec une relation contenant tous les attributs (même les attributs hérités) de la classe
   - En cas de classe concrète avec des classes filles : Clé primaire des relations correspondantes aux classes filles = clés étrangères faisant référence à la clé primaire de la relation correspondant à la classe mère

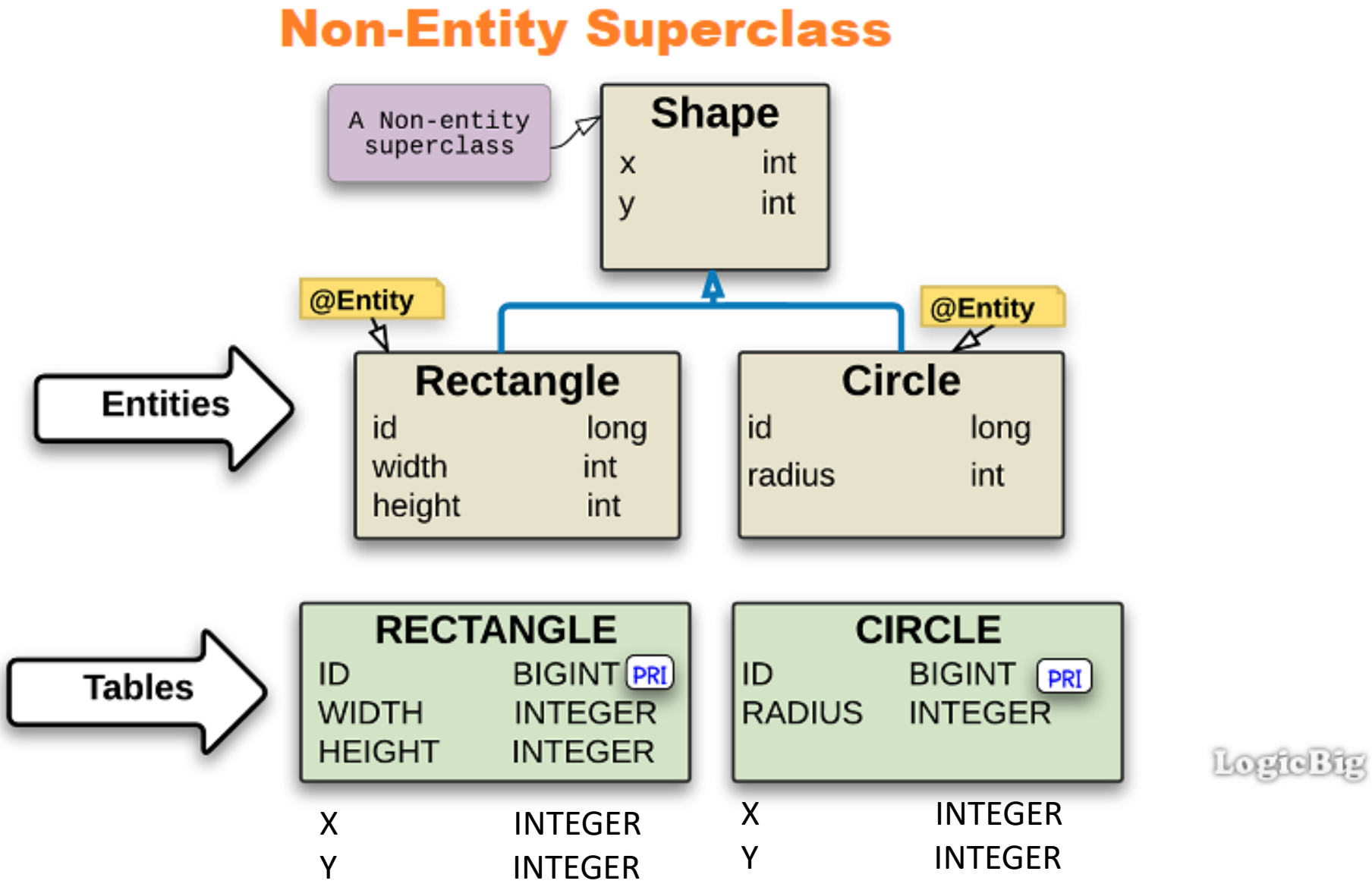
 Pas de jointure pour retrouver les informations

 Problème pour les associations et requêtes polymorphes

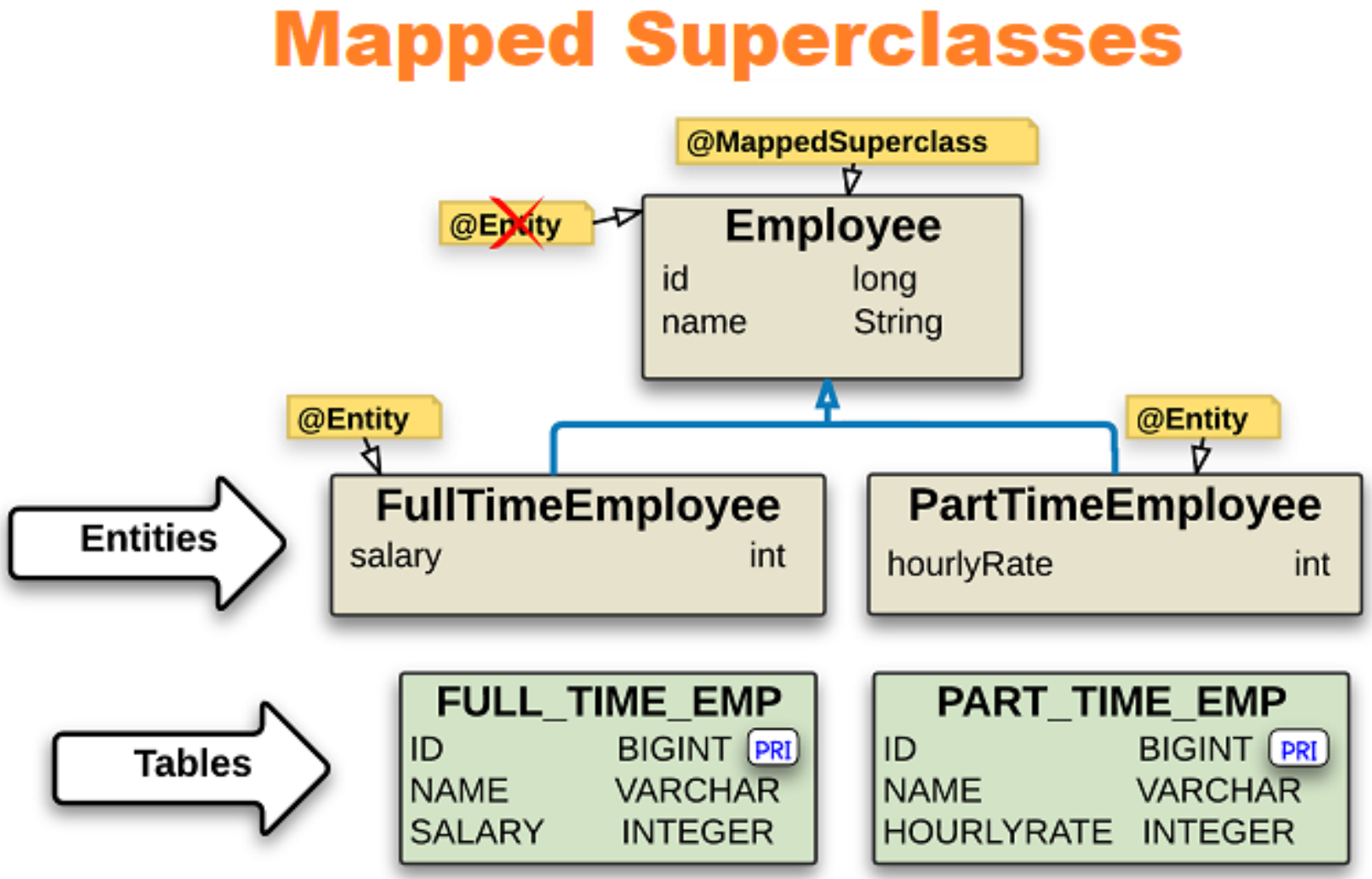
 Redondance d'information

   - Pas de nécessité d'offrir cette solution dans les spec. JPA

Annotation JPA : une relation/classe concrète pour une hiérarchie contenant une classe abstraite

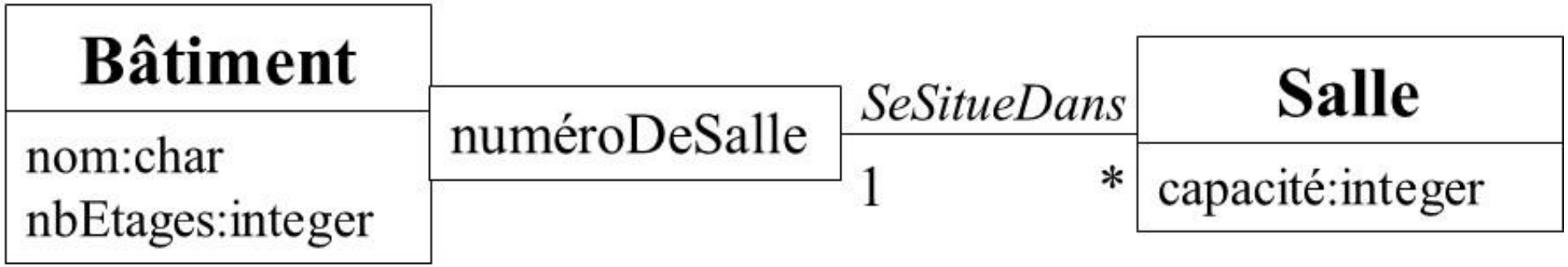


Annotation JPA : une relation/classe concrète pour une hiérarchie ne contenant pas de classe abstraite



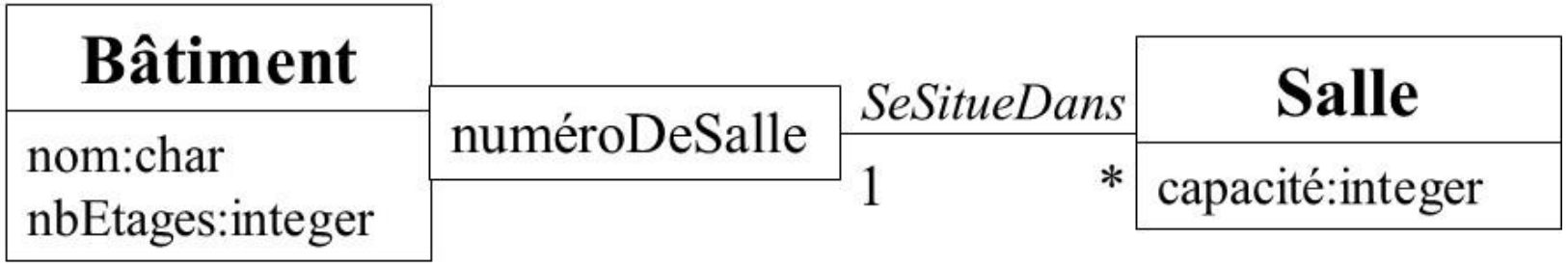
Objet dépendant (1/2) : principes

- Objet dont le cycle de vie dépend d'un autre objet (objet propriétaire)

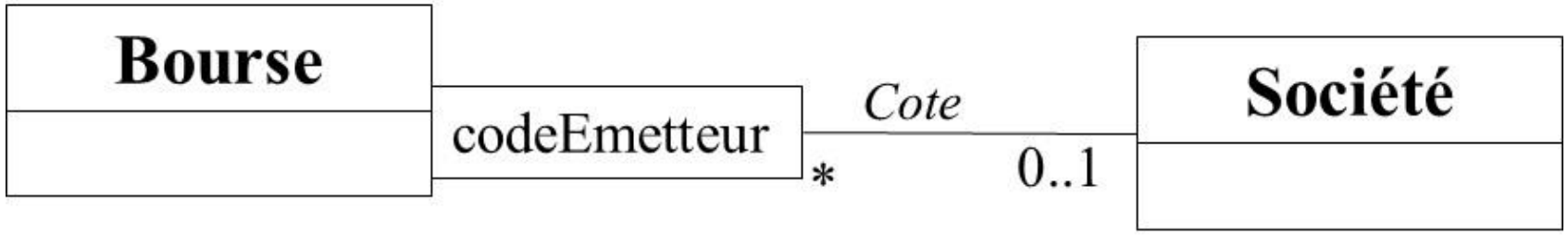


- Clé primaire de la relation correspondant à la classe des objets dépendants constituée de la clé primaire de de la relation correspondant à la classe des objets propriétaires
- Persistance d'un objet dépendant gérée par l'objet propriétaire (ou par le *DAO* de l'objet propriétaire)

Objet dépendant (2/2) : exemples



Batiment(IDBatiment, nom, nbEtages)
Salle(#IDBatiment, numeroSalle, capacité)



Bourse (IDBourse, …)
Societe(IDSociete, …)
Cotation(#IDBourse, codeEmetteur, #IDSociete)

Navigation dans le graphe d'objets

Que faire lorsqu'un objet est créé à partir de données récupérées dans la base de données ?

- Récupération immédiate et création des objets associés

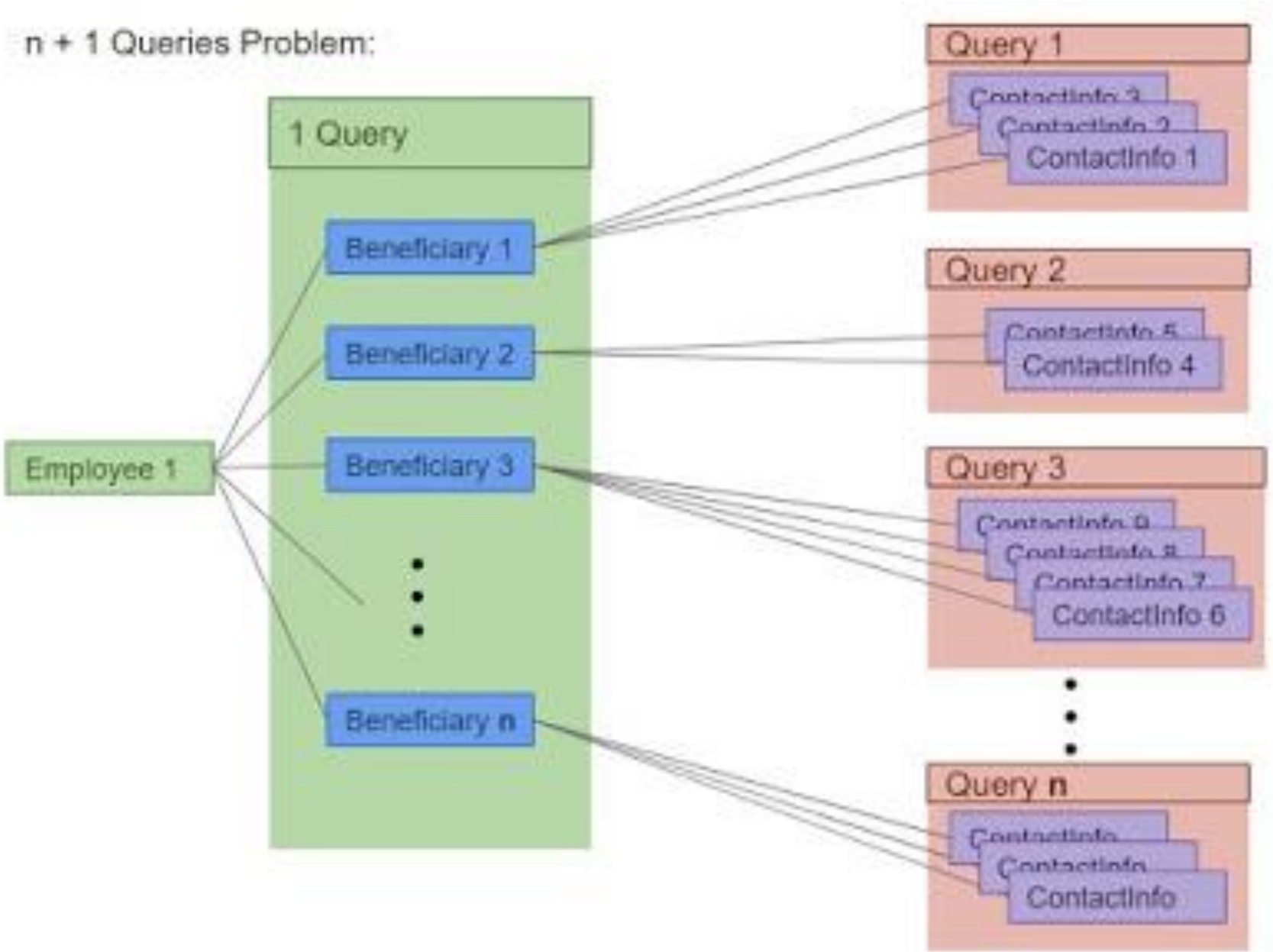
  Risque de récupération d'objets inutiles et mauvaises performances sans raison valable

- Création des objets associés uniquement en cas d'accès par l'application (*lazy loading* ou récupération paresseuse ou à la demande)

  Problème des « N+1 Selects »

- Choix du type de récupération en fonction de la requête

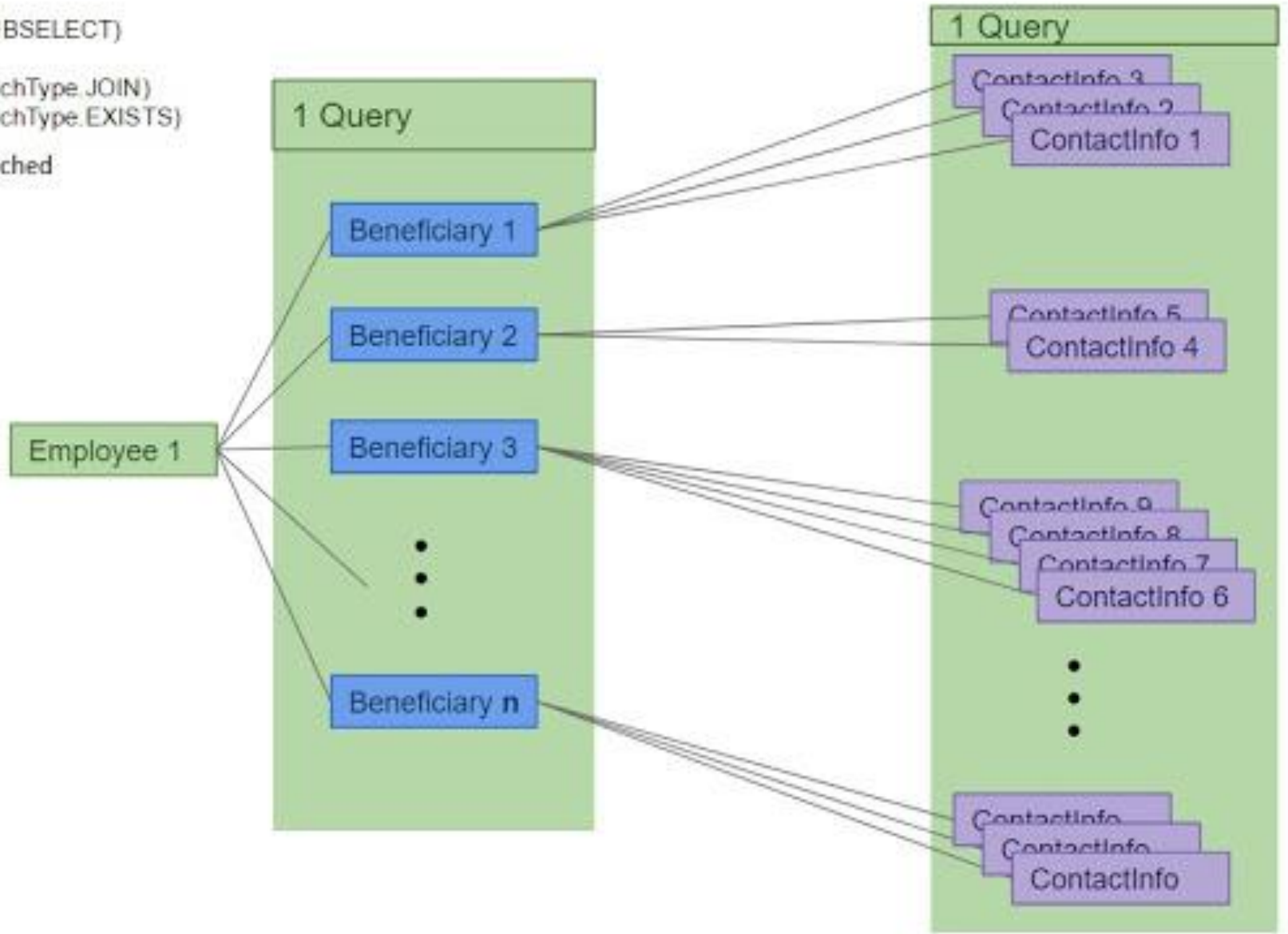
Problème des N+1 SELECT



Chargement : par sous-SELECT

Hibernate:
@Fetch(FetchMode.SUBSELECT)
Eclipselink:
@BatchFetch(BatchFetchType.JOIN)
@BatchFetch(BatchFetchType.EXISTS)

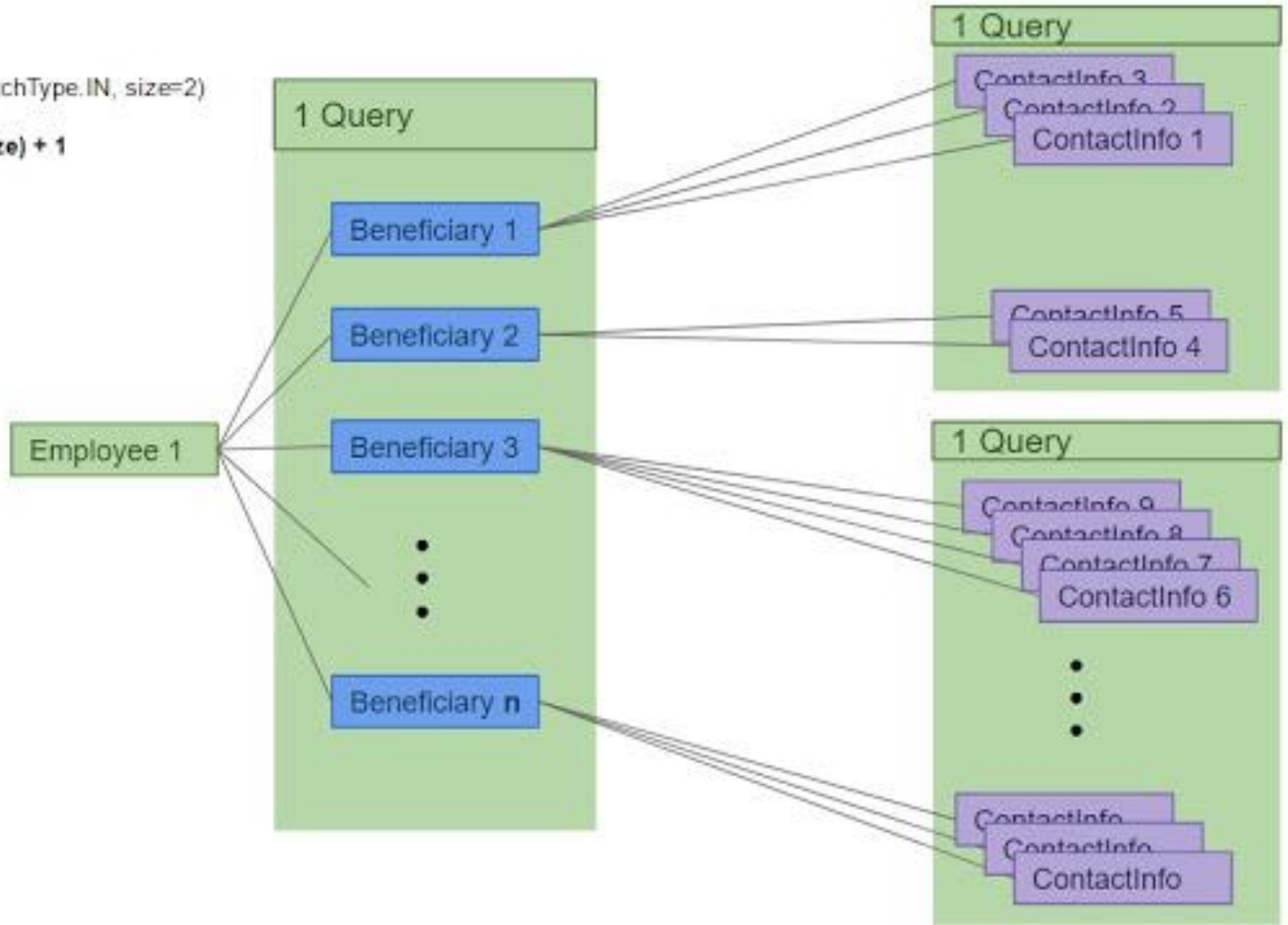
Now, n = # of *associations* batched



Chargement : par sous-SELECT par paquet

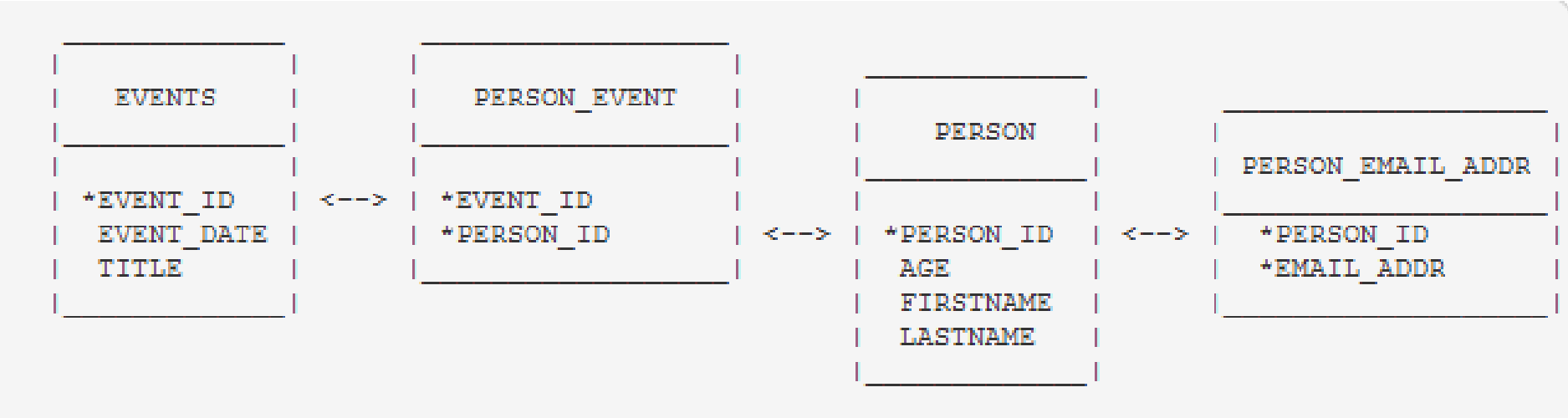
Hibernate:
@BatchSize(2)
Eclipselink:
@BatchFetch(BatchFetchType.IN, size=2)

Now, **n+1** becomes **(n/batchSize) + 1**



Exemple issu du tutoriel Hibernate

Modèle relationnel :



Modèle objet :

```
public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;
    private Set events ;
    private Set emailAddresses ;
    …
}
```

```
public class Event {

    private Long id;
    private String title;
    private Date date;
    private Set participants;
    …
}
```

Tutoriel Hibernate : Exemples d'annotations

Exemple de la classe Person :

```
@Entity                                  ← Classe event.Person
@Table (name="PERSON")                   « mappée » en relation PERSON
public class Person {
    private Long id;
    private int age;
    private String firstname;
    private String lastname;
    public Person() {}

    // Accessor methods
    @Id@GeneratedValue(strategy=GenerationType.AUTO)   } Mapping de l'identifiant
    @Column(name="PERSON_ID")
    public Long getId() { return this.id; }

    private Set events = new HashSet();
    @ManyToMany          ← Mapping de collection d'objets
    @JoinTable( name="PERSON_EVENT",
                joinColumns=@JoinColumn(name="PERSON_ID"),
                inverseJoinColumns=@JoinColumn(name="EVENT_ID") )
    protected Set getEvents() { return events;}

    private Set emailAddresses = new HashSet();

    @CollectionOfElements    ← Mapping de collection de valeurs
    @JoinTable( table=@Table(name="PERSON_EMAIL_ADDR"),
                joinColumns = @JoinColumn(name="PERSON_ID") )
    public Set getEmailAddresses() { return emailAddresses; }
}
```

Tutoriel Hibernate : exemples de nuplets

Modèle relationnel :

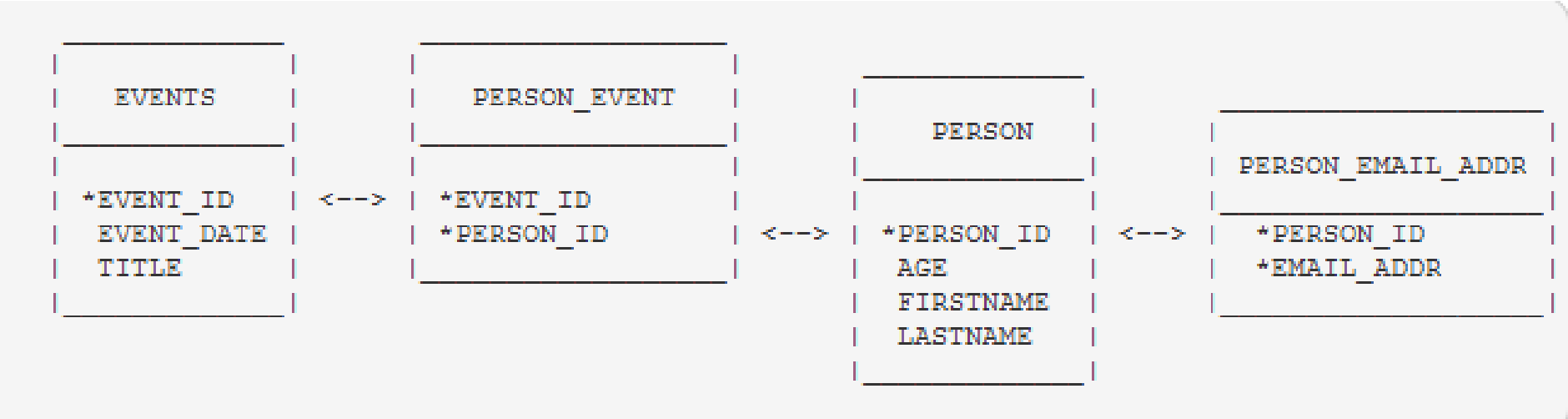


Table PERSON_EVENT

	event_id [PK] bigint	person_id [PK] bigint
1	3	2
2	6	2
3	3	4
4	3	5
5	7	5
6	8	5

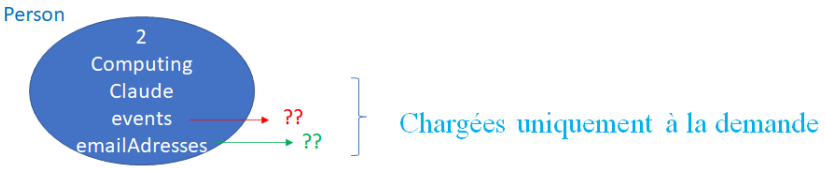
Tutoriel Hibernate : Exemple de génération de requêtes

- **Chargement tardif (*lazy*) par défaut pour les deux collections `events` et `participants` :**

```
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
```

⇒ **`select … from PERSON person0_ where person0_.PERSON_ID=?`**

Récupération du nuplet correspondant à la personne 2



- **Chargement agressif (*eager*) pour la collection `events` avec stratégie par défaut (`select`):**

```
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
```

⇒ **`select … from PERSON person0_ where person0_.PERSON_ID=?`**

⇒ **`select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?`**

Chargement tardif (*lazy*) par défaut pour `participants`

Récupération du nuplet correspondant à la personne 2 et des événements associés à la personne 2, i.e. les évènements 3 et 6

- **Chargement agressif pour la collection `events` avec stratégie charg. par jointure (`join`) :**

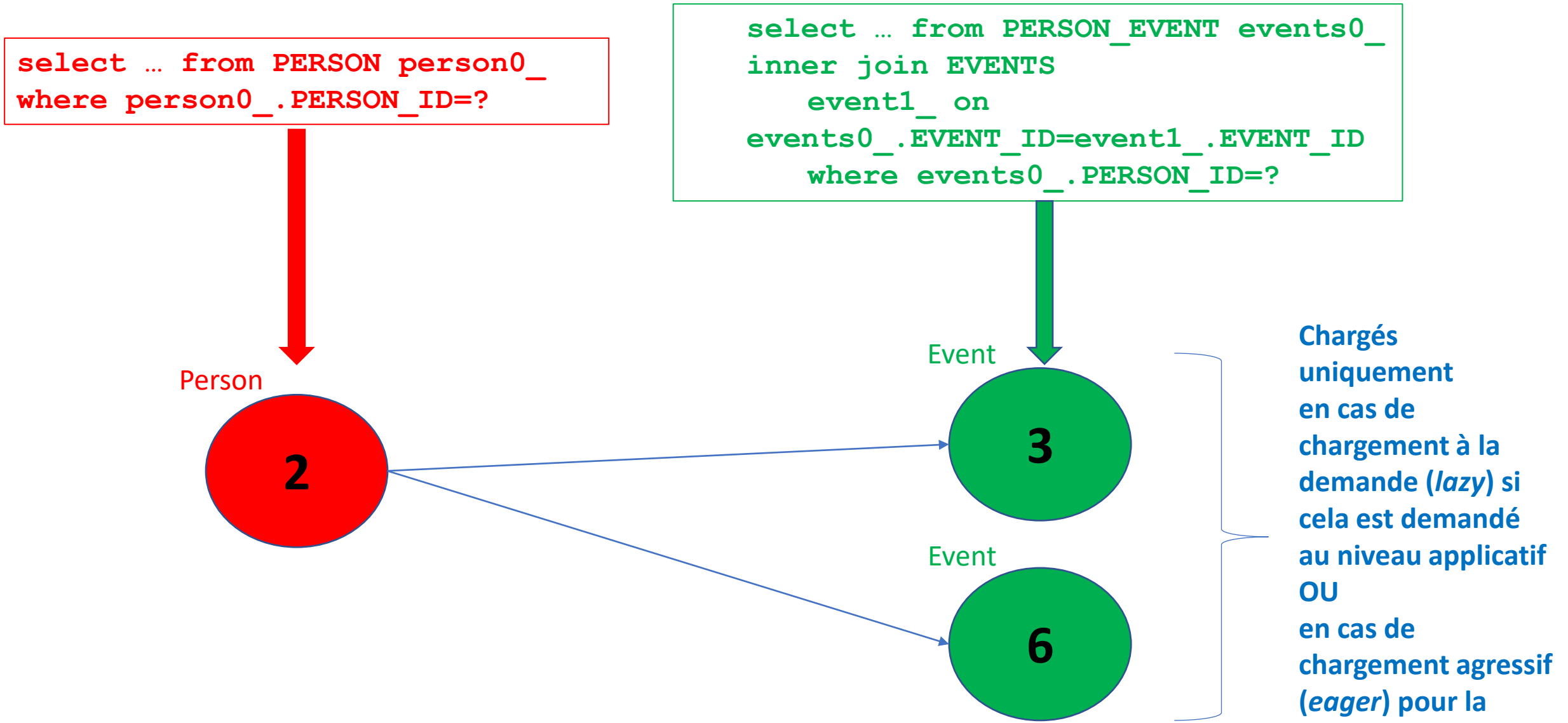
```
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
```

⇒ **`select … from PERSON person0_ left outer join PERSON_EVENT events1_ on person0_.PERSON_ID=events1_.PERSON_ID left outer join EVENTS event2_ on events1_.EVENT_ID=event2_.EVENT_ID where person0_.PERSON_ID=?`**

Chargement tardif (*lazy*) par défaut pour `participants`

Récupération du nuplet correspondant à la personne 2 et des événements associés à la personne 2, i.e. les évènements 3 et 6

Tutoriel Hibernate : Exemple de graphe d'objets chargés en mémoire



Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (1/10)

```
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
```

① `select … from PERSON person0_ where person0_.PERSON_ID=?`

② `select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?`

③ `select … from PERSON_EVENT participan0_ inner join PERSON person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where participan0_.EVENT_ID=?`

2 fois car 2 événements au lieu du sous select!! Une fois pour l'événement 3 et une fois pour le 6

④ `select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?`

⑤ `select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?`

⑥ `select … from PERSON_EVENT participan0_ inner join PERSON person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where participan0_.EVENT_ID=?`

2 fois car 2 événements au lieu du sous select!! Une fois pour l'événement 7 et une fois pour le 8

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (2/10)

Table PERSON_EVENT

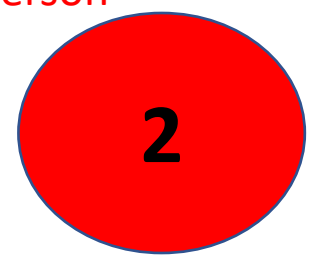
	event_id [PK] bigint	person_id [PK] bigint
1	3	2
2	6	2
3	3	4
4	3	5
5	7	5
6	8	5

① Récupération du nuplet correspondant à la personne 2
② Récupération des événements associés à la personne 2 : 3 et 6
③ Récupération des personnes associées aux événements associés à la personne 2 : (3,2) ; (3,4); (3,5) et (6,2) => *2 requêtes, une requête / événement*
④ Récupération des événements associés à la personne 4 : il n'y a que 3
⑤ Récupération des événements associés à la personne 5 : 3, 7 et 8
⑥ Récupération des personnes associées aux événements associés à la personne 5 : aucun => *2 requêtes, une requête / événement*

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (3/10)

Après l'exécution de la requête 1 :

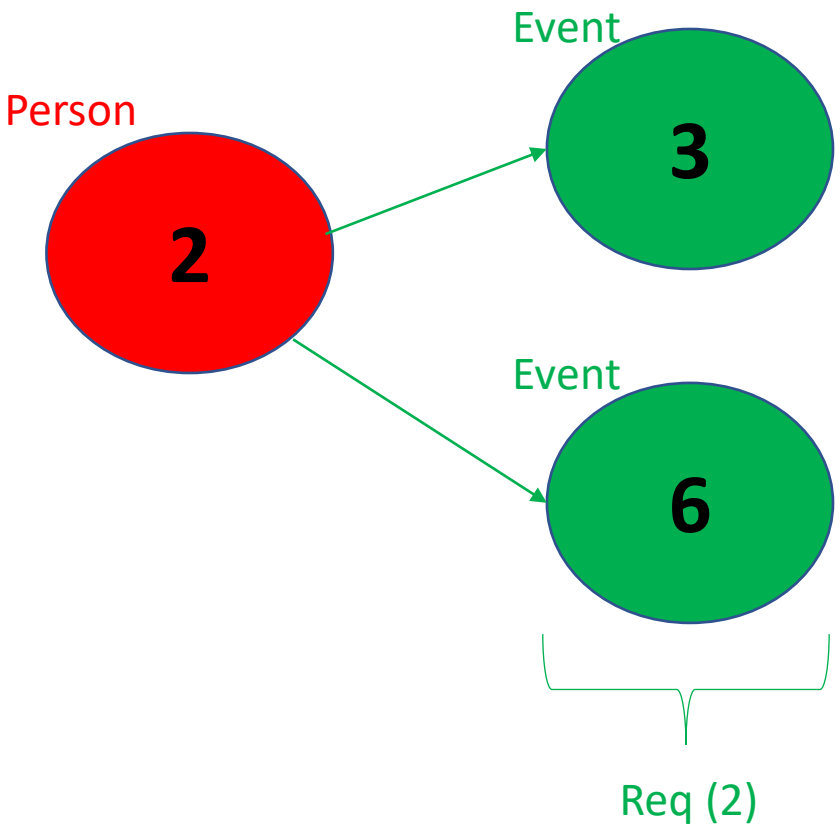
Person



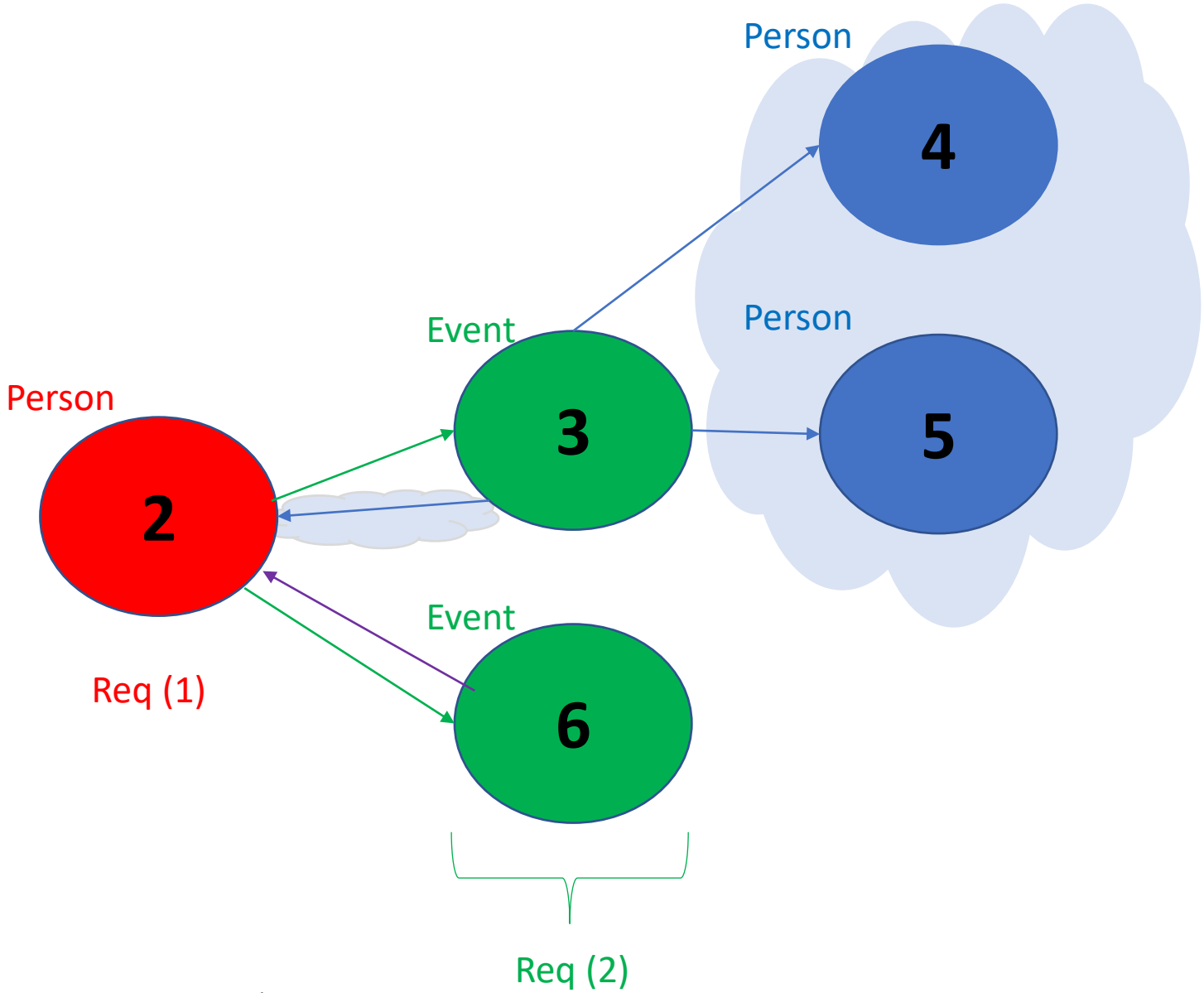
Req (1)

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (4/10)

Après l'exécution de la requête 2 :

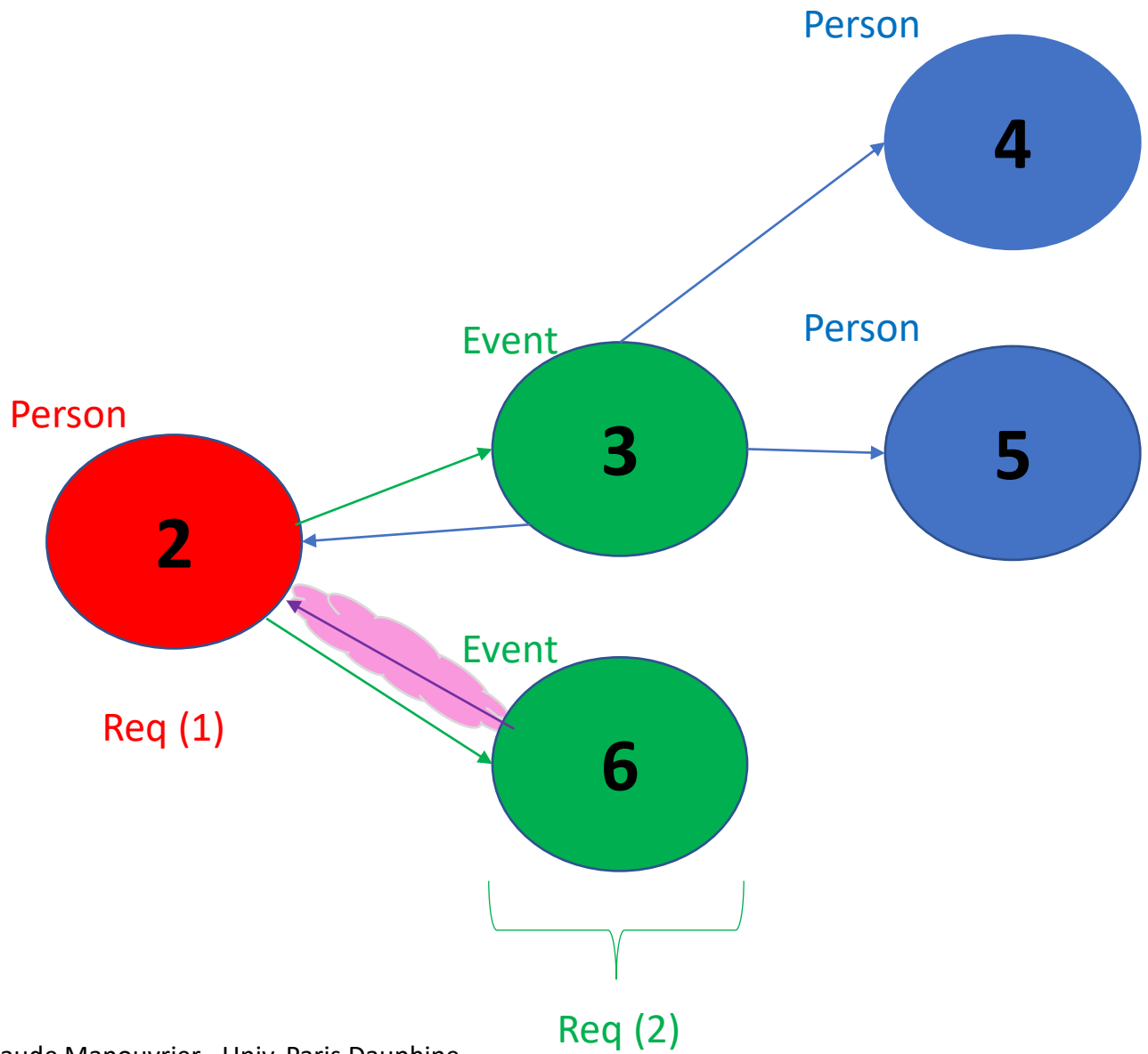


Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (5/10)



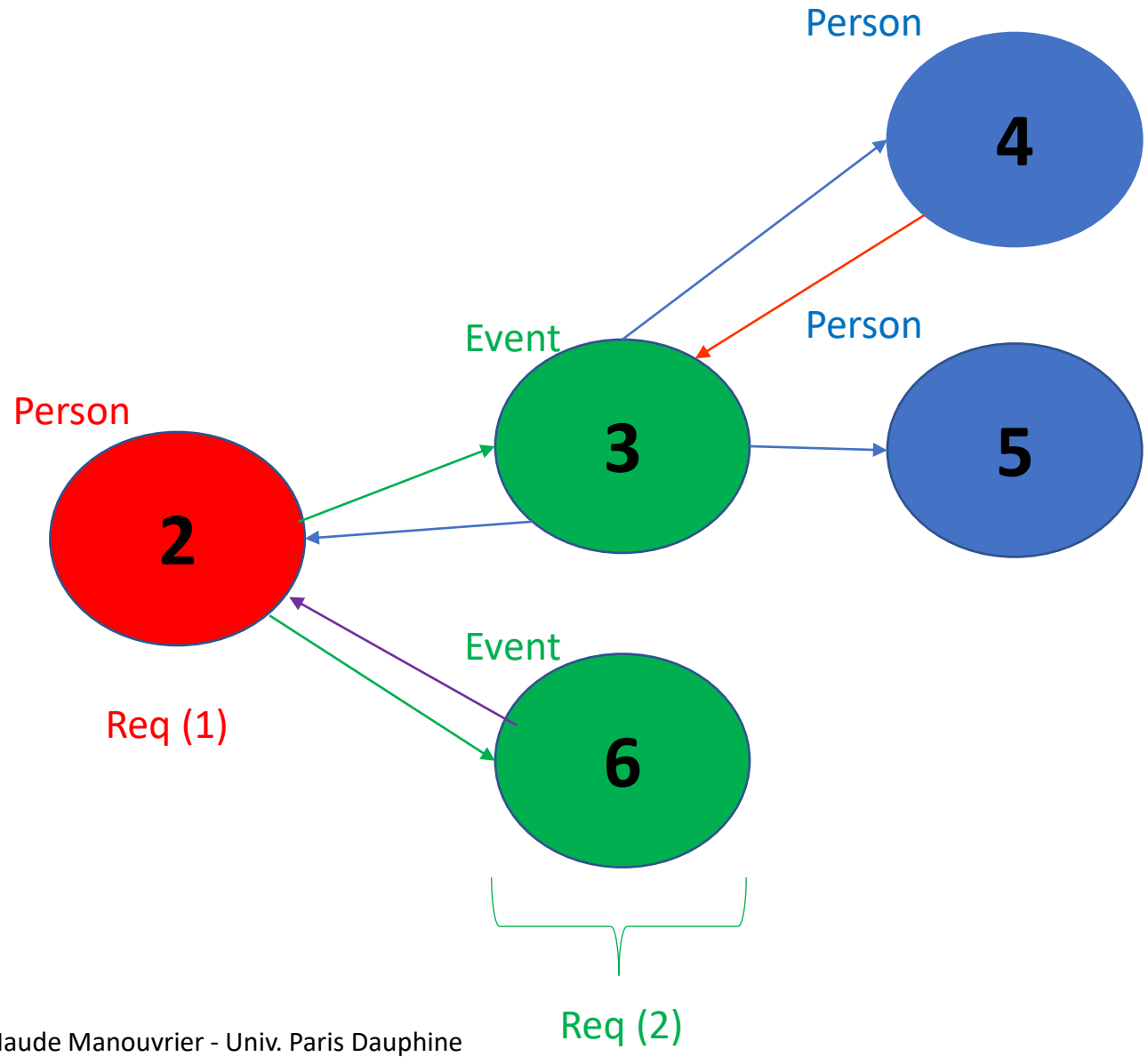
Après l'exécution de la requête 3 pur l'événement 3

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (6/10)



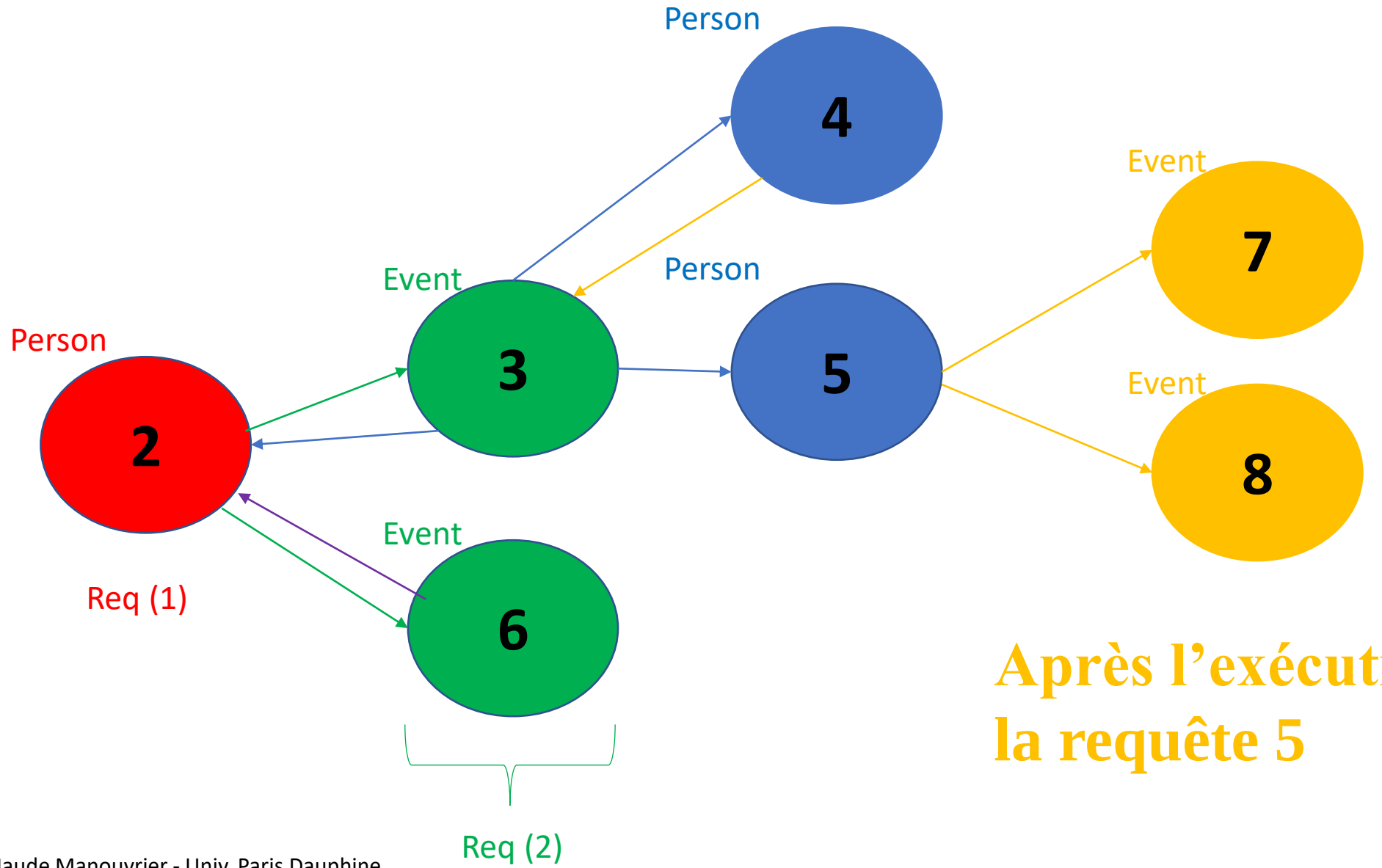
Après l'exécution de la requête 3 pur l'événement 6

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (7/10)



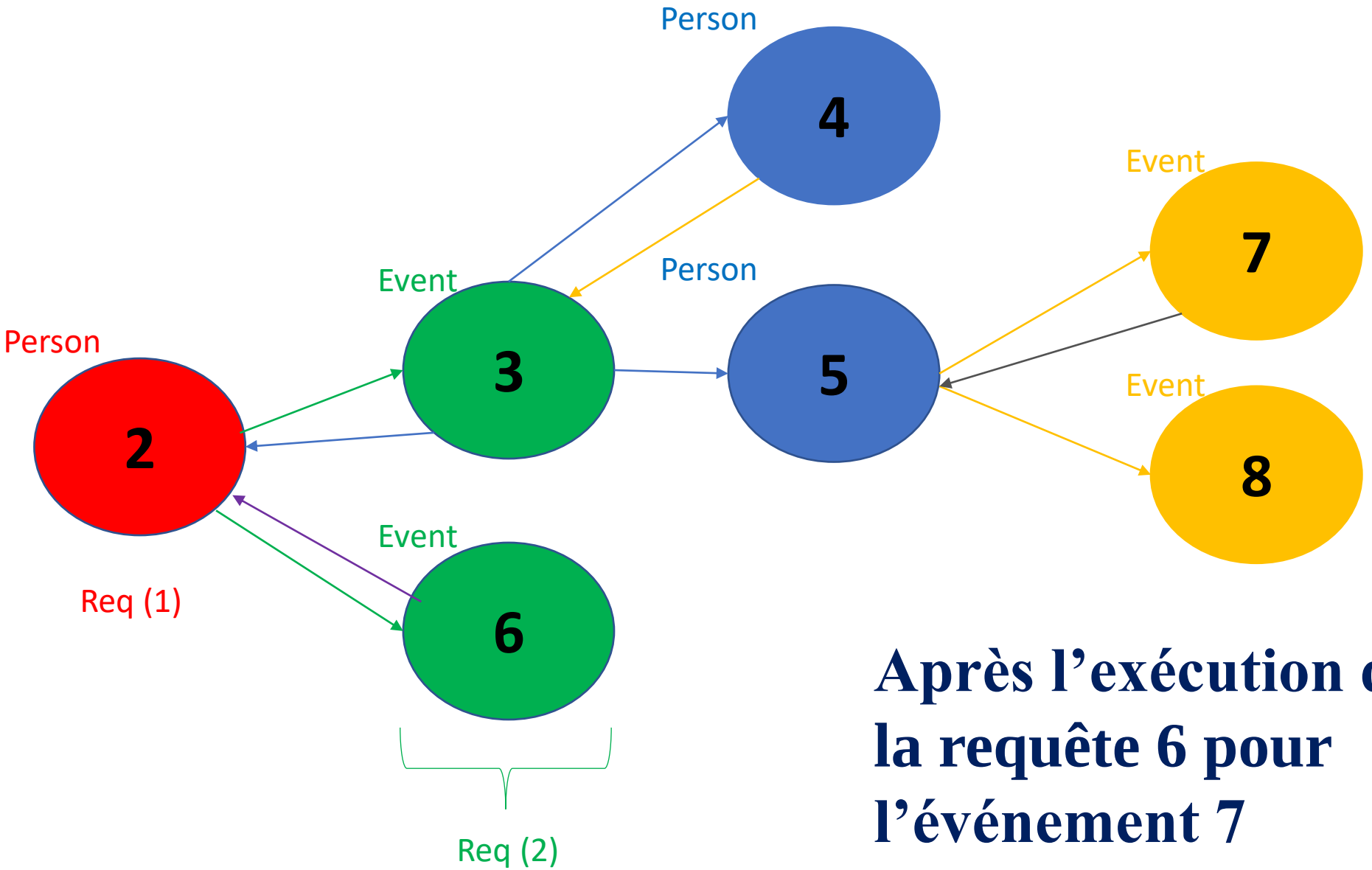
Après l'exécution de la requête 4

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (8/10)



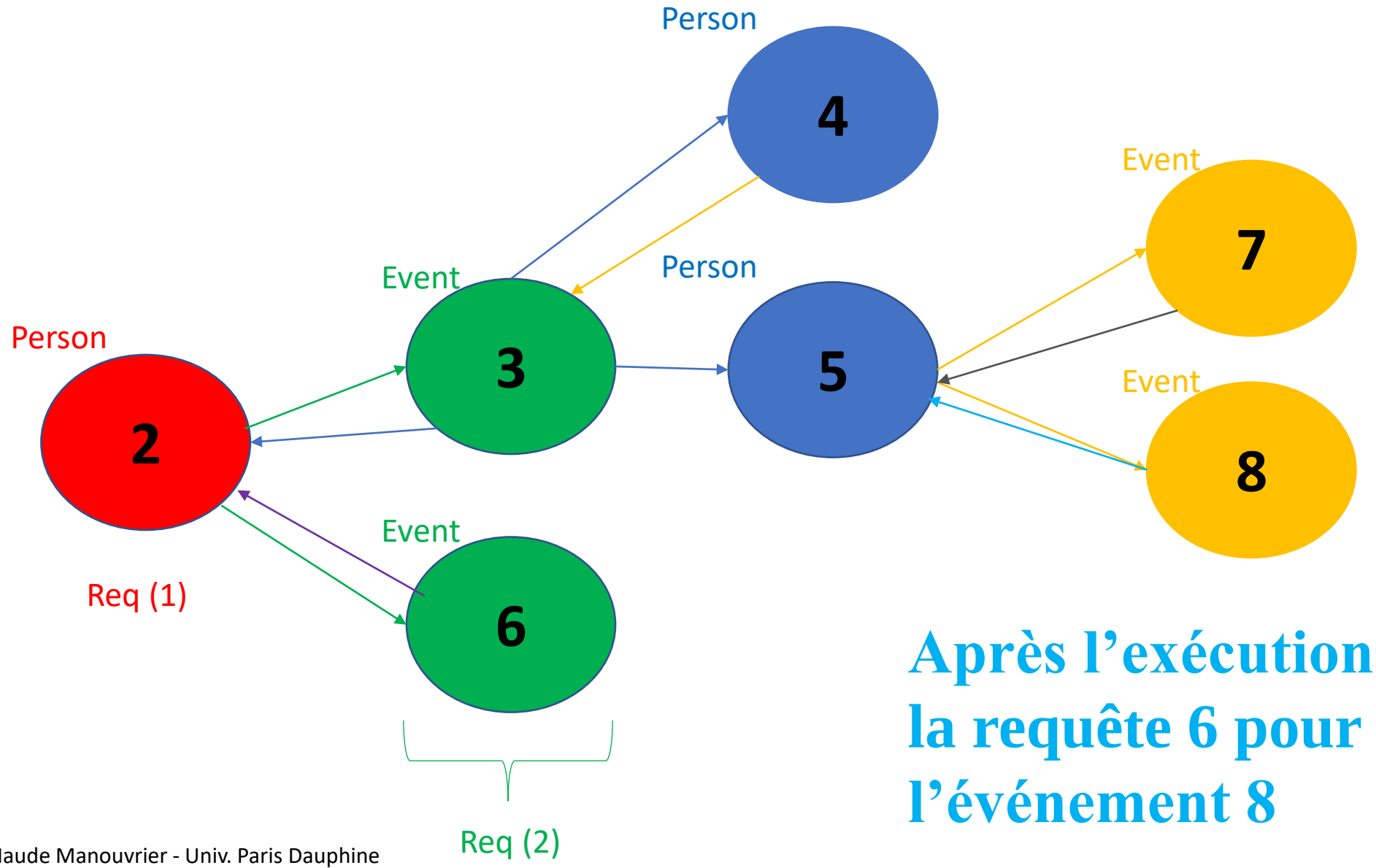
Après l'exécution de la requête 5

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (9/10)



Après l'exécution de la requête 6 pour l'événement 7

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par select pour la coll. Participants (10/10)



Après l'exécution de la requête 6 pour l'événement 8

Tutoriel Hibernate : Charg. agressif pour les 2 collections et avec stratégie charg. par sous-select pour la coll. Participants

```
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
```

```
① select … from PERSON person0_ where person0_.PERSON_ID=?
② select … from PERSON_EVENT events0_ inner join EVENTS
   event1_ on events0_.EVENT_ID=event1_.EVENT_ID
   where events0_.PERSON_ID=?
③ select … from PERSON_EVENT participan0_ inner join PERSON
   person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where
   participan0_.EVENT_ID in (select event1_.EVENT_ID from
   PERSON_EVENT events0_ inner join EVENTS event1_ on
   events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?)
④ select … from PERSON_EVENT events0_ inner join EVENTS event1_
   on events0_.EVENT_ID=event1_.EVENT_ID
   where events0_.PERSON_ID=?
⑤ select  … from PERSON_EVENT events0_ inner join EVENTS event1_
   on events0_.EVENT_ID=event1_.EVENT_ID
   where events0_.PERSON_ID=?
⑥ select … from PERSON_EVENT participan0_ inner join PERSON
   person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where
   participan0_.EVENT_ID in (select event1_.EVENT_ID from
   PERSON_EVENT events0_ inner join EVENTS event1_ on
   events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?)
```

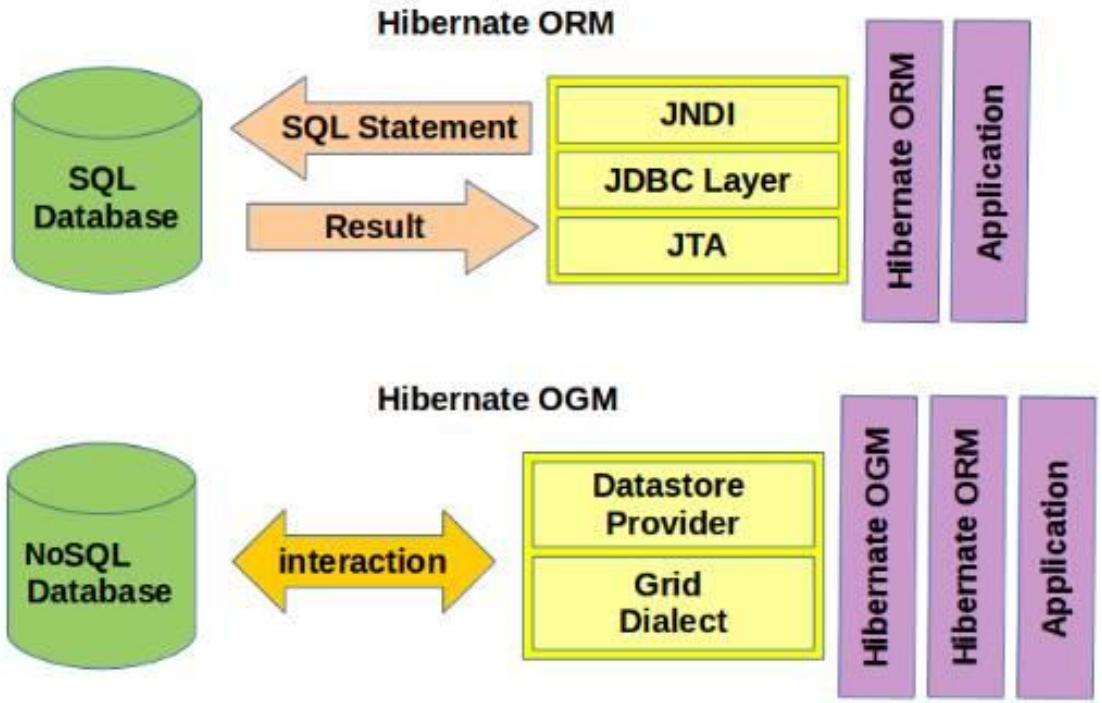
Récupération via une seule req. des participants associés aux événements associés à la personne 2

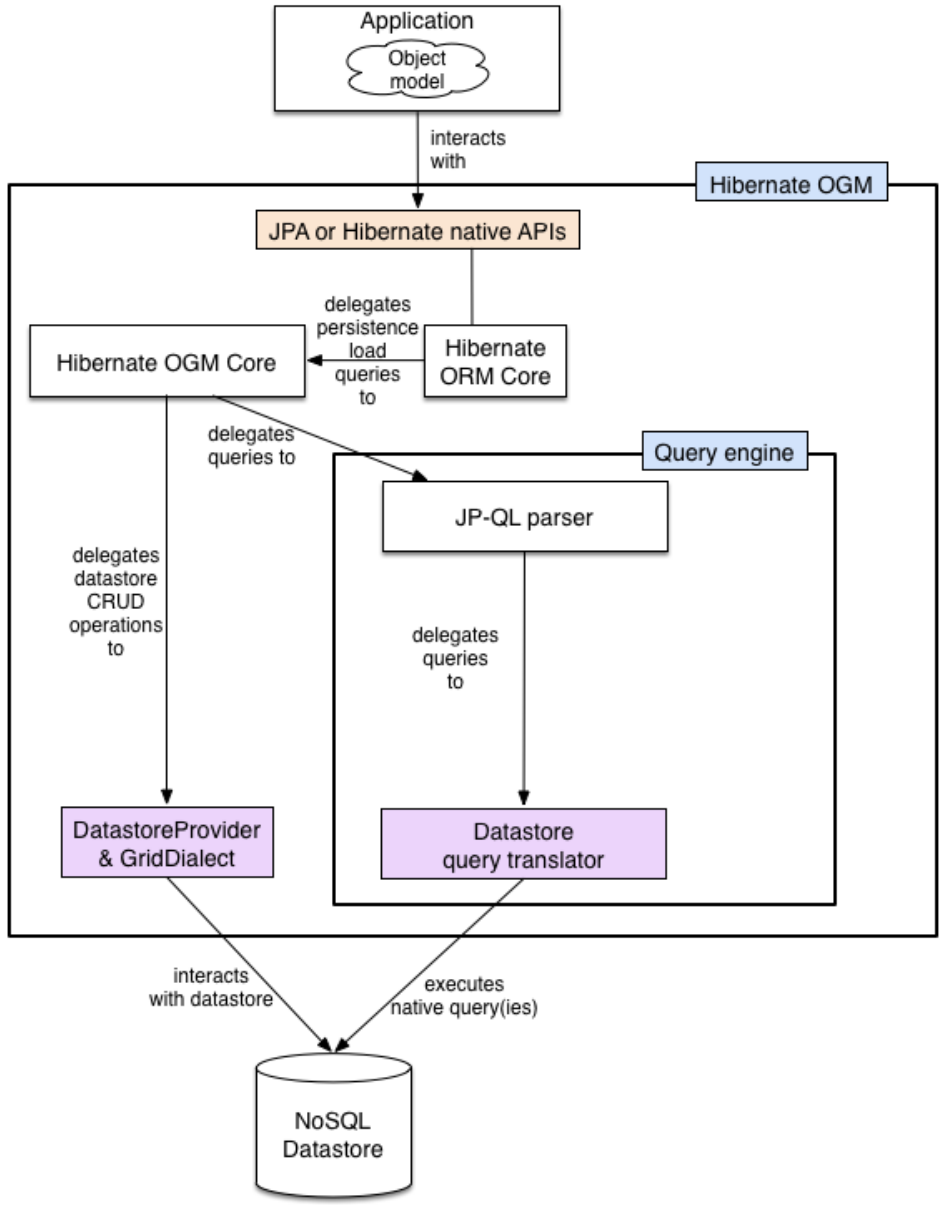
Récupération via une seule req. des participants associés aux événements associés à la personne 5

ORM pour le NoSQL

- **Hibernate Object/Grid Mapper (OGM)** : https://hibernate.org/ogm/
  - Clé-valeur : Infinispan
  - Document : MongoDB
  - Graphe : Neo4j
  - + Cassandra, Redis, CouchDB (expérimental - développement par la communauté)
- **Eclipse JNoSQL** : https://www.jnosql.org/
  - Implémentation de référence de Jakarta NoSQL
  - Couche de communication (Diana) : qui fournit un ensemble d'API conçues pour définir la communication avec les bases de données NoSQL
  - une couche de correspondance (Artemis) : qui fournit un ensemble d'API permettant aux développeurs d'intégrer des applications Java à des bases de données NoSQL
  - Fonctionne avec plusieurs moteurs NoSQL

Hibernate OGM





Eclipse JNoSQL