Langage Java

Persister des données avec Java Persistence API

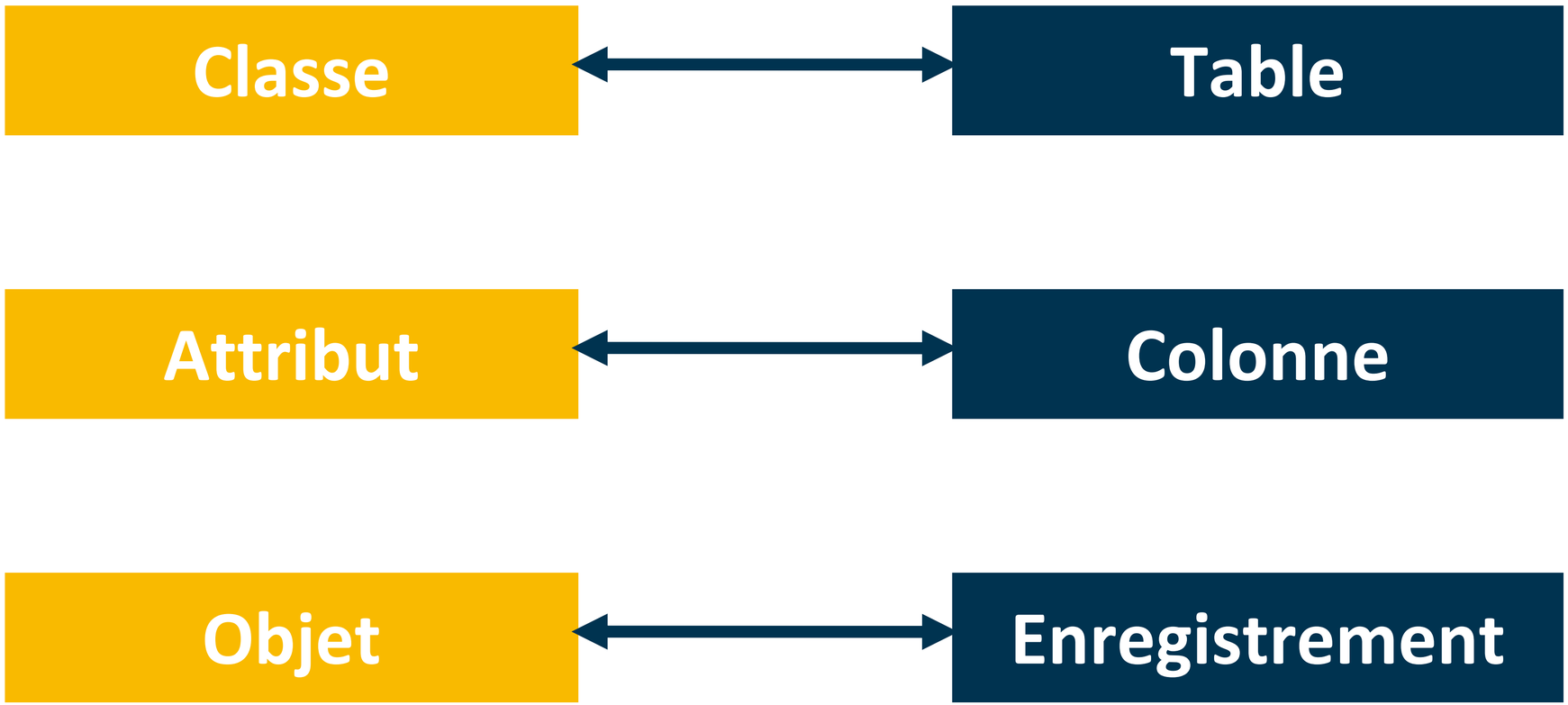
Chapitre 1
JPA - Introduction

Sommaire

Couche de persistance sans ORM

- Le développeur a la possibilité d'écrire manuellement la persistance des objets:
  - Toutes les étapes d’utilisation de JDBC apparaissent clairement dans le code des classes
  - La gestion des connexions
  - L’écriture manuelle des requêtes
  - Plus de dix lignes de codes sont nécessaires pour rendre persistante une classe contenant des propriétés simples
  - Développement objets "très couteux"

ORM - Principe



ORM - Exemple

```
public class Personne {
        private int id;
        private String nom;
        private String prenom;
}

Personne p1 = new Personne(1, « Durand », « Paul »);
Personne p2 = new Personne(2, « Dupont », « Pierre »);
```

Table PERSONNE

ID	NOM	PRENOM
1	Durand	Paul
2	Dupont	Pierre

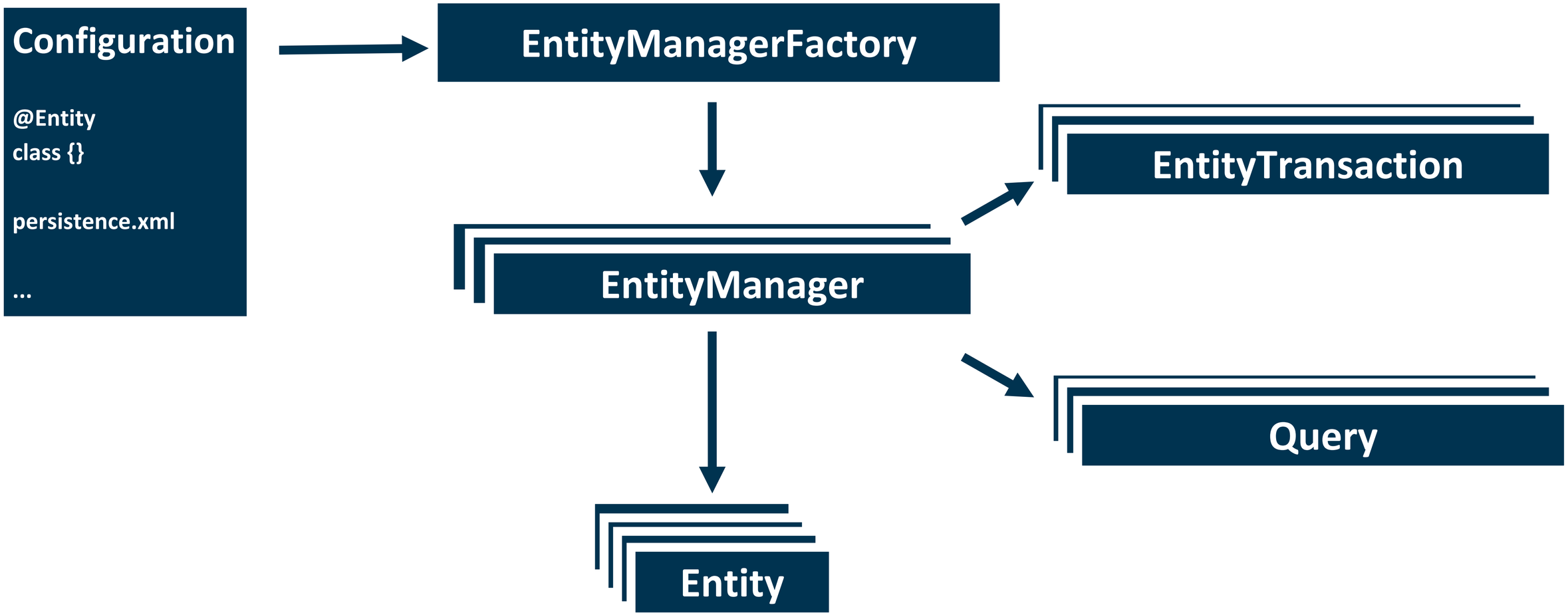
ORM – Un peu d'histoire

- 1994 : TopLink, premier ORM au monde, en smallTalk
- 1996 : TopLink propose une version Java
- 1998 : EJB 1.0
- 2000 : EJB 2.0
- 2001 : Lancement d'Hibernate (alternative aux EJB 2)
- 2003 : Gavin King, créateur d'Hibernate, rejoint JBoss
- 2006 : EJB 3.0 et JPA 1.0
- 2007 : TopLink devient EclipseLink
- 2009 : JPA 2.0
- 2013 : JPA 2.1

JPA = Java Persistence API

- Les entités persistantes sont des POJO
- Les entités ne se persistent pas elles-mêmes : elle passent par l'**EntityManager**
- L'EntityManager se charge de persister les entités en s'appuyant sur les annotations qu'elles portent
- La classe **Query** encapsule les résultats d'une requête
- Le langage **JPQL** : (Java Persistence Query Language). Une requête JPQL est analogue à une requête SQL sur des objets

Les objets JPA



Configuration (persistence.xml) (1/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd>

        <persistence-unit name="nantes-jpa" transaction-type="RESOURCE_LOCAL">
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                <properties>
                        <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:8889/nantes-bdd"/>
                        <property name="javax.persistence.jdbc.user" value="nantes_user"/>
                        <property name="javax.persistence.jdbc.password" value="nantes_pass"/>
                        <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
                </properties>
        </persistence-unit>

</persistence>
```

RESOURCE_LOCAL => utilisation en dehors d'un conteneur Java EE

Configuration (persistence.xml) (2/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
        version="2.0">
    <persistence-unit name="manager1" transaction-type="JTA">
        <jta-data-source>java:/DefaultDS</jta-data-source>
        <mapping-file>ormap.xml</mapping-file>
        <class>fr.nantes.Employee</class>
        <class>fr.nantes.Person</class>
        <class>fr.nantes.Address</class>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
        </properties>
    </persistence-unit>
</persistence>
```

JTA => utilisation dans conteneur Java EE

Mapping JPA

- JPA doit faire un pont entre le monde relationnel de la base de données et le monde objet

- Ce pont est fait par configuration :

  - Avec des fichiers XML. C'était quasiment l'unique façon de faire jusqu'à l'avènement du JDK 1.5

  - Avec des annotations Java depuis le JDK 1.5

Les entités

- Le bean entity est composé de propriétés mappées sur les champs de la table de la base de données.
- Ce sont de simples POJO (Plain Old Java Object).
- Un POJO n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.

Conditions pour les classes entités

- Un bean entité doit obligatoirement
  - Avoir un constructeur **sans argument**
  - Être déclaré avec l'annotation **@javax.persistence.Entity**
  - Posséder au moins une propriété déclarée comme clé primaire avec l'annotation **@Id**

Exemple d'entité

```java
@Entity
@Table(name="personne")
public class Personne {


  @Id
  private Integer id;

  @Column(name = "NOM", length = 30, nullable = false, unique = true)
  private String nom;

  @Column(name = "PRENOM", length = 30, nullable = false)
  private String prenom;

  // constructeur sans argument
  public Personne() {
  }

  // getters and setters
}
```

@Entity Indique que la classe doit être gérée par JPA

@Table (facultatif) désigne la table à mapper

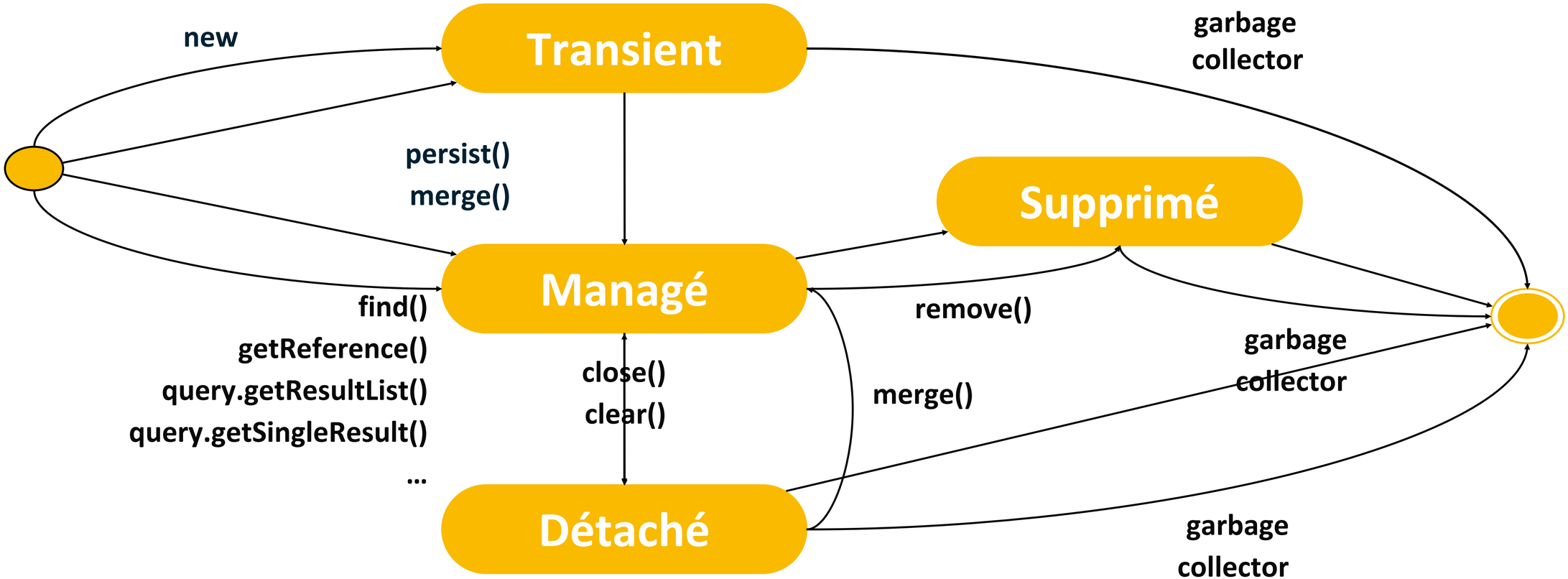
@Id Clé primaire

@Column
pour Associer un champ à une colonne de la table

Chapitre 2
JPA - CRUD

Sommaire

Cycle de vie d'une entité



EntityManager

```
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("nantes-jpa");
EntityManager em = entityManagerFactory.createEntityManager();
```

- Qu'est ce que **nantes-jpa** ?
  - La classe **Persistence** charge le fichier **persistence.xml**
  - Dans ce fichier, il y a une configuration nommée nantes-jpa

EntityManager

EntityManagerFactory entityManagerFactory = **Persistence**.*createEntityManagerFactory("***nantes-jpa***");*

EntityManager em = entityManagerFactory.createEntityManager();

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd>

    <persistence-unit name="nantes-jpa" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/nantes-bdd"/>
            <property name="javax.persistence.jdbc.user" value="nantes_user"/>
            <property name="javax.persistence.jdbc.password" value="nantes_pass"/>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
        </properties>
    </persistence-unit>

</persistence>
```

Atelier (TP)

OBJECTIFS : Extraire des informations d'une base de données

DESCRIPTION :

- Dans le TP n°1 vous allez vous connecter à une base de données et en extraire des informations.

Insérer une donnée

- La méthode persist (…) permet de demander l'enregistrement d'une **entité JPA**

```
EntityManager em = entityManagerFactory.createEntityManager();

Hotel h = new Hotel();
h.setNom("BeauRegard");
h.setVille("Lisbonne");
em.persist(h);
```

- Pour récupérer l'identifiant généré (ex: 13465):
  - récupérer l'ID depuis l'objet « sauvé »

```
long id = h.getId();
```

Récupérer une instance

- La recherche par la clé primaire est faite avec la méthode :
  - **find(…, …)**

Si dans la classe Hotel, la clé primaire est du type **int** alors le second argument est de type **int**

```
Hotel h = em.find(Hotel.class, 120);
if (h != null){
        //traitement
}
```

  - Renvoie **null** si l'occurrence n'est pas trouvée

Récupérer une instance

- La recherche par requête est réalisée grâce aux:
  - Langage de requêtes JPQL
  - Méthodes **createQuery()** de EntityManager

```
TypedQuery<Hotel> query2 = em.createQuery("select h from Hotel h where h.nom='nom2'",
Hotel.class);

Hotel h2 = query2.getResultList().get(0);
```

Modifier une instance

- Pour modifier un objet il faut
  - Le lire
    Récupération de l'objet en lecture/écriture
  - Le modifier
    Aucune méthode particulière n'est invoquée lors de la modification d'un objet !

```
Hotel h = em.find(Hotel.class, 120);
If (h != null){
        h.setNom("Beauregard");
}
```

Ce que vous ne pouvez pas faire…

- Supposons qu'il y ait un Hotel d'identifiant 2 en base de données et que je souhaite le mettre à jour.
  - Cas 1: je créé l'instance à la main, elle est donc Transient, dans ce cas il faut la pousser dans le contexte de persistance avec un merge.

```
Hotel hotel = new Hotel();
hotel.setId(2);
hotel.setNom("Untel");
hotel.setVille("nouvelle ville");
em.merge(hotel);
```

- Il faut d'abord charger l'hotel dans le **contexte de persistence** avant de le mettre à jour:

```
Hotel hotel = em.find(Hotel.class, 2);
hotel.setNom("Untel");
hotel.setVille("nouvelle ville");
```

Contexte de persistence

- Tous les objets chargés via une requête ou la méthode find sont mis dans le persistence context et sont au statut "Managé".
- L'entité ci-dessous n'a pas été créée par Hibernate (via un find ou TypedQuery):

```
Hotel hotel = new Hotel();
hotel.setId(2);
hotel.setNom("Untel");
hotel.setVille("nouvelle ville");
```

- Elle est "Transient".

Merger les données

```
Hotel hotel = new Hotel();
hotel.setId(hotel.getId());
hotel.setNom(hotel.getNom());
hotel.setVille("nouvelle ville");
em.merge(hotel);
```

- Permet de mettre à jour un objet "détaché" ou "transient". Le merge fait passé le statut de l'objet de "détaché" à "managé", puis le met à jour.
  - Entité détachée: chargée depuis un entityManager fermé entre temps.

Supprimer les données

```
Hotel h = em.find(Hotel.class, 120);

If (h != null){
   em.remove(h);
}
```

- L'objet supprimé **doit faire partie du contexte de persistance**.
- Il faut d'abord l'extraire de la base pour le supprimer.

Mapping Clé Primaire (1)

- @Id : il permet d'associer un champ de la table à la propriété en tant que clé primaire

- @GeneratedValue : indique que la clé primaire est générée via une stratégie. Contient plusieurs attributs :
  - Strategy : Précise le type de générateur à utiliser :
    - *TABLE,*
    - *SEQUENCE,*
    - *IDENTITY*
    - *AUTO. La valeur par défaut est AUTO*
  - Generator : nom du générateur à utiliser

Générateurs d'identifiants

- Attention, les stratégies de génération de valeur de clé primaire ne marche pas sur toutes les bases de données.

- Les types possibles sont les suivants :
  - AUTO: laisse la base de données choisir la stratégie de génération par défaut pour générer la valeur de la clé primaire. Exemple: IDENTITY pour MySQL.
  - **IDENTITY: utilise une colonne pour générer la clé primaire (AUTO_INCREMENT pour MySQL).**
  - TABLE: utilise une table dédiée qui stocke les clés des tables générées. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @javax.persistence.TableGenerator
  - SEQUENCE: utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @javax.persistence.SequenceGenerator

Exemple

```
@Entity
@Table(name="LIVRE")
public class Livre {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
}
```

Exemple pour un ID en AUTO_INCREMENT (MySQL ou MariaDB)

- Evidemment la colonne ID en base doit avoir l'option AUTO_INCREMENT.

Autres annotations basiques

@Temporal: Indique quel type est le type SQL d'une colonne / champ de type date / heure.

TemporalType.DATE: désigne une date seule sans heure associée

TemporalType.TIME: désigne une heure

TemporalType.TIMESTAMP: désigne une date avec une heure

@Transient : Indique de ne pas tenir compte du champ lors du mapping

@Enumerated : utiliser pour mapper des énumération de type entier ou chaine de caractères

Java 8 : mapper les LocalDate et LocalDateTime

- **@Temporal** ne fonctionne pas avec les classes LocalDate et LocalDateTime.
- 2 choses à faire:
  - **@Column** doit être utilisée sur les attributs de type LocalDate et LocalDateTime
  - Il faut ajouter la librairie de compatibilité Hibernate pour Java 8
  - Attention: elle doit être dans la même version que la librairie **hibernate-entitymanager**

Exemple:

```
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-java8</artifactId>
	<version>5.4.10.Final</version>
</dependency>
```

Transaction avec JPA

```
EntityTransaction et = em.getTransaction();
et.begin();


...
em.persist(p1);
...
em.persist(p2);
...
et.commit();       // ou et.rollback()
em.close();
```

Atelier (TP)

OBJECTIFS : Echanger des informations avec une base de données

DESCRIPTION :

- Dans le TP n°2 vous allez mettre en place votre première entité.

Chapitre 3
JPA – Les relations

Sommaire

Relations

@JoinColumn

Relation 1-n

Relation n-1

Relation n-n

Relation 1-1

Relations

- Une relation peut-être unidirectionnelle ou bidirectionnelle.

- Les associations entre entités correspond à une jointure entre tables de la base de données:
  - Les tables sont liées grâce à une contrainte de clé étrangère

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

- Règle :
  - @JoinColumn : définit la colonne de jointure.
  - @JoinTable : définit la table de jointure

@JoinColumn

Sans cette annotation le nom est défini par défaut:
<nom_entité>_<clé_primaire_entité>

Attributs :

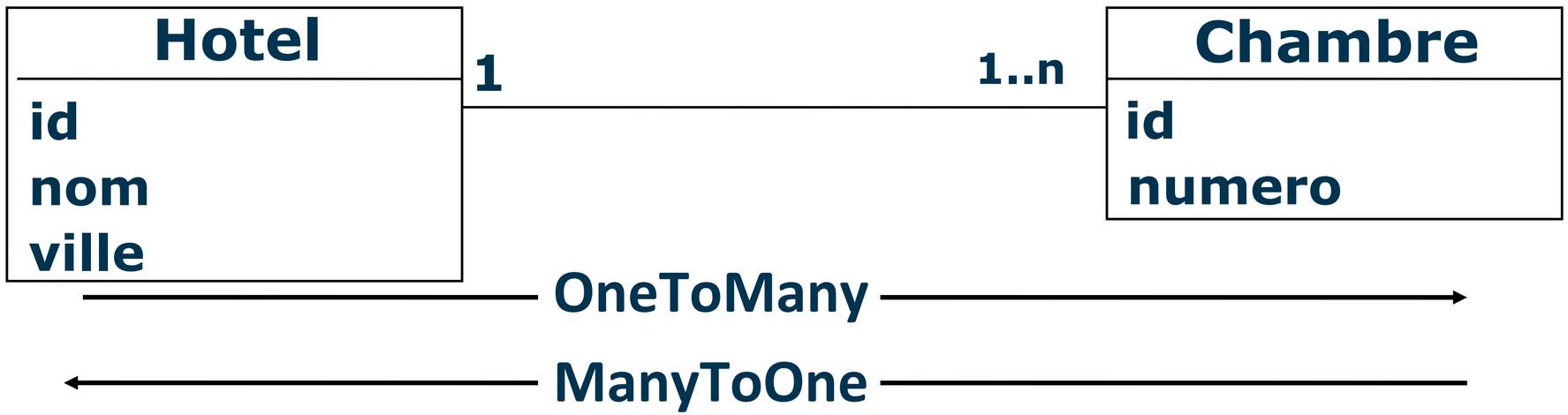
- name
  - *Nom de la colonne correspondant à la clé étrangère.*
- referencedColumnName
  - *Nom de la clé étrangère.*
- unique
  - *Permet de définir la contrainte d'unicité de l'attribut. (Par défaut, false)*
  - *S'ajoute aux contraintes définies avec l'annotation @Table.*
- nullable
  - *Définit si une colonne accepte la valeur nulle. (Par défaut, true)*

Relation 1-n

Exemple

Une Chambre appartient à un Hôtel

Un Hôtel regroupe plusieurs Chambres



@ManyToOne

```
@Entity
@Table (name=« CHAMBRE")
public class Chambre {

   @Id
    private int id;

   @Column(name="NUMERO")
    private int numero;

    @ManyToOne
    @JoinColumn(name="HOT_ID")
    private Hotel hotel;

}
```

@ManyToOne associé à l'annotation @JoinColumn

@OneToMany

```
public class Hotel {
   private long id;

   @OneToMany(mappedBy="hotel")
   private Set<Chambre> chambres; // référence vers les chambres

   public Hotel() {
     chambres = new HashSet<Chambre>();
   }
   …
}
```

@OneToMany porte l'attribut mappedBy

Récapitulatif : Relation 1-n / n-1


```
public class Hotel {


   @OneToMany(mappedBy="hotel")
   private Set<Chambre> chambres;
}
```

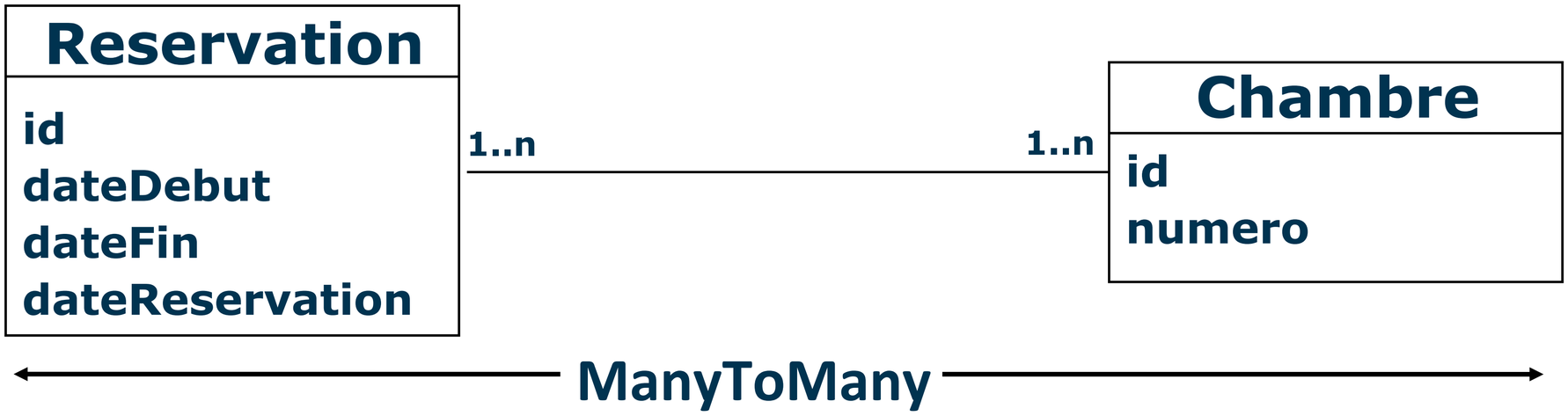
```
public class Chambre {

   @ManyToOne
   @JoinColumn(name="HOT_ID")
   private Hotel hotel;
}
```

Relation n-n

Exemple

Une Reservation peut concerner plusieurs Chambre

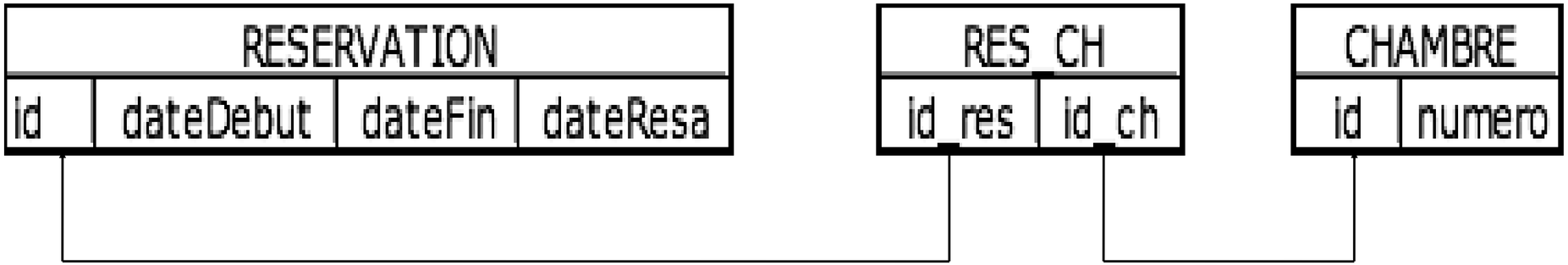
Une Chambre peut faire l'objet de plusieurs Reservation



Relation n-n

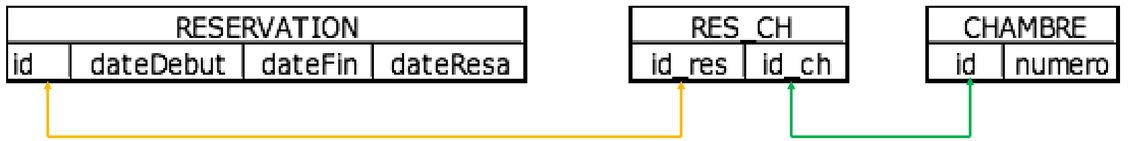
En base de données, utilisation d'une table intermédiaire

D’un point de vue du MCD, correspond à 2 relations 1:n



@ManyToMany

```
@Table (name="RESERVATION")
public class Reservation {
        @Id
        @Column(name="ID")
        private int id;

        @ManyToMany
        @JoinTable(name="RES_CH",
                joinColumns= @JoinColumn(name="ID_RES", referencedColumnName="ID"),
                inverseJoinColumns= @JoinColumn(name="ID_CH", referencedColumnName="ID")
        )
    private Set<Chambre> chambres;
}
```



La propriété **joinColumns** décrit la jointure entre la table « RESERVATION » et la table RES_CH

La propriété **inverseJoinColumns** décrit la jointure entre la table « CHAMBRE » et la table RES_CH

@ManyToMany

```
@Table (name="CHAMBRE")
public class Chambre {

        @ManyToMany(mappedBy="chambres")
        private Set<Reservation> reservations;
}
```

Dans ce cas la classe Chambre est esclave.
Si on déclare @JoinTable dans les 2 classes, aucune classe n'est maître ou esclave.

Relation 1-1

@OneToOne par clé étrangère

```
public class Facture {
        @Id
        private int id;

        @OneToOne
        private Reservation reservation;  // référence vers la réservation
        ...
        public Reservation getReservation() { ... }
        public void setReservation(Reservation res) { ...}
}
```

Atelier (TP)

OBJECTIFS : Utiliser des annotations

DESCRIPTION :

- Dans le TP n°3 vous allez mettre en place des relations au niveau des entités.

Chapitre 4
Embeddable et embedded

@Embeddable, Embedded

PERSONNES

ID	NOM	PRENOM	NUMERO	RUE	VILLE

```
@Entity
@Table(name="PERSONNES")
public class Personne {

    private Long id;
    private String nom;
    private String prenom;

    @Embedded
    private Adresse adresse;

}
```

1 Table => 2 classes.

```
@Embeddable
public class Adresse {
        private Integer numero;
        private String rue;
        private String ville;
}
```

Chapitre 5
L'héritage

Sommaire

Héritage

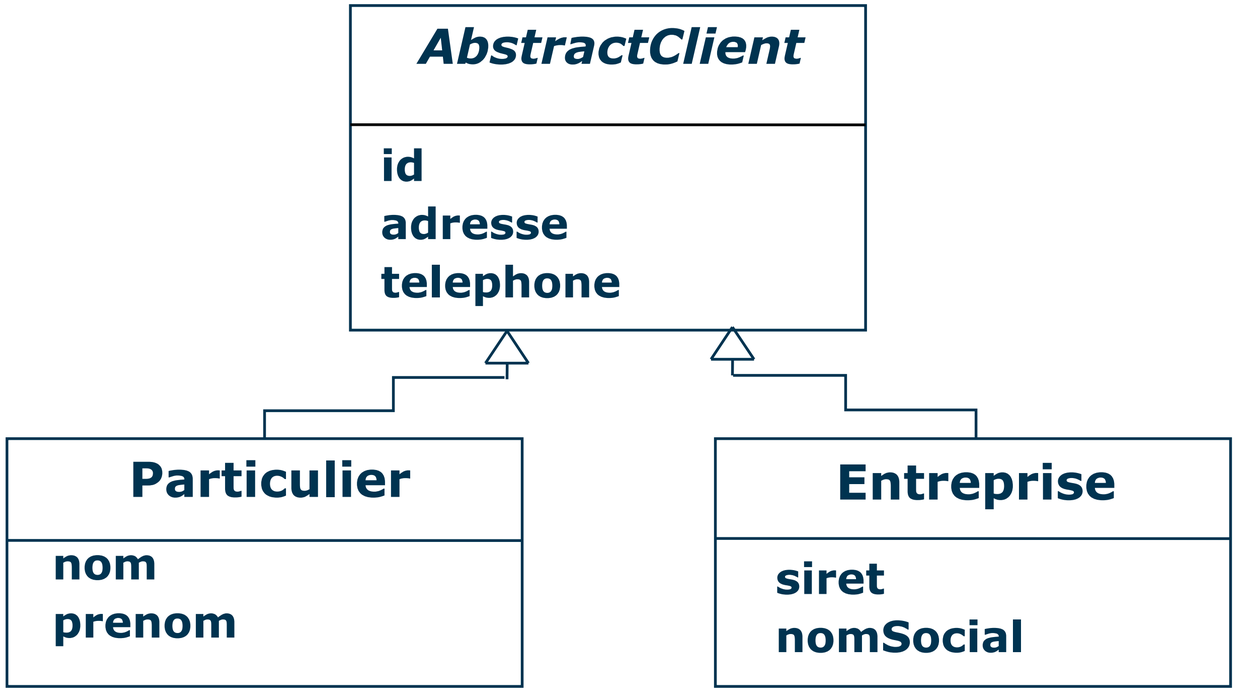
- L’héritage est l’un des 3 grands principes de l’objet parmi:
  - Encapsulation
  - Héritage
  - Polymorphisme

- 3 stratégies permettent le mapping O/R de l’héritage
  - Chaque stratégie a ses avantages et ses inconvénients

- JPA permet l’utilisation de ces 3 stratégies générales

Héritage : Exemple

- Un Client peut être
  - soit un Particulier
  - soit une Entreprise



1 table par classe

- Appellation: « Table per subclass »

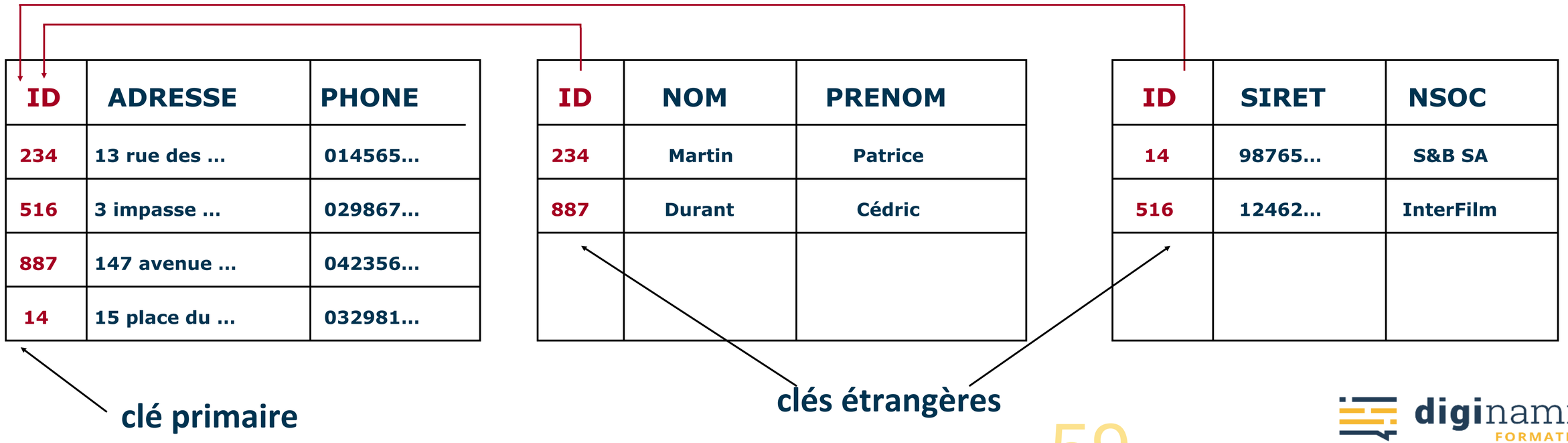
- Idée générale
  - A chaque classe du modèle objet correspond une table
  - 3 tables dans l'exemple CLIENT-PARTICULIER-ENTREPRISE
  - Chaque table contient les attributs de l'objet + l'identifiant
  - L'héritage entre classes est modélisé par des clés étrangères (qui sont généralement aussi clés primaires)

- Conceptuellement
  - (+) Solution simple et efficace

- Techniquement
  - (-) Jointures lors de chaque requête à performances non optimales

1 table par classe

- 1 table pour l'objet AbstractClient
- 1 table pour l'objet Particulier
- 1 table pour l'objet Entreprise
- 2 clés étrangères pour représenter les deux relations d'héritage



1 table par classe

```java
/**
 * Client abstrait: Entreprise ou particulier.
 */
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class AbstractClient {

	@Id
	@GeneratedValue
	private Integer id;
	...
}
```

```java
/**
 * Modélise une entreprise.
 */
@Entity
public class Entreprise extends AbstractClient {

	private String nomSocial;
	private String siret;
	…
```

```java
/**
 * Un particulier.
 */
@Entity
public class Particulier extends AbstractClient {

	private String nom;
	private String prenom;
	…
```

1 table par classe

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasse() {
        Entreprise e = new Entreprise(); // création d'une entreprise
        e.setAdresse("1");
        e.setTelephone("0141220300");
        e.setNomSocial("SQLi");
        e.setSiret("44019981800012");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transaction = em.getTransaction();
        transaction.begin();
        em.persist(e);

              Particulier p = new Particulier(); // création d'un particulier
        p.setAdresse("14 rue de la Foret");
        p.setTelephone("0112341234");
        p.setNom("Durant");
        p.setPrenom("Cédric");
        em.persist(p);

              transaction.commit();
        Entreprise e2 = (Entreprise) em.find(AbstractClient.class, 3);
        Particulier p2 = (Particulier) em.find(AbstractClient.class, 4);
        LOGGER.debug("Client: " + p2);
        LOGGER.debug("Client: " + e2);
}
```

1 table pour toute la hiérarchie

- Appellation: « SINGLE_TABLE »
- Idée générale
  - Toutes les informations de toutes les classes sont stockées dans une seule et unique table
  - Toutes les colonnes ne servent pas à toutes les classes, seules certaines colonnes sont utilisées par chaque classe
  - Chaque enregistrement est « typé » avec un indicateur permettant de retrouver le type de l'instance → discriminateur (discriminator)
- Conceptuellement
  - (+) Simple à mettre en place
  - (-) Une solution faible, pas très évolutive
- Techniquement
  - (-) « NOT-NULL » inutilisable sur les colonnes (problèmes d'intégrité)
  - (-) Des enregistrements quasiment vides (espace perdu)
  - (+) Pas de clés étrangères, pas de jointure au requêtage

1 table pour toute la hiérarchie

- Une seule table dans laquelle sont stockées
  - ET les informations des Particuliers
  - ET les informations des Entreprises

- La colonne TYPE contient le discriminateur
  - P → Particulier E → Entreprise

ID	TYPE	ADRESSE	PHONE	NOM	PRENOM	SIRET	NSOC
234	P	13 rue des …	014565…	Martin	Patrice	*null*	*null*
516	E	3 impasse …	029867…	*null*	*null*	12462…	InterFilm
887	P	147 avenue …	042356…	Durant	Cédric	*null*	*null*
14	E	15 place du …	032981…	*null*	*null*	98765…	S&B SA

clé primaire

1 table pour toute la hiérarchie

```java
/**
 * Client abstrait: Entreprise ou particulier.
 */
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE")
public class AbstractClient {

	@Id
	@GeneratedValue
	private Integer id;
	...
}
```

Nom de la colonne servant de discriminant

```java
/**
 * Modélise une entreprise.
 *
 */
@Entity
@DiscriminatorValue("E")
public class Entreprise extends AbstractClient {

	private String nomSocial;
	private String siret;
	…
```

```java
/**
 * Un particulier.
 *
 */
@Entity
@DiscriminatorValue("P")
public class Particulier extends AbstractClient {

	private String nom;
	private String prenom;
	…
```

1 table pour toute la hiérarchie

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasse() {
		Entreprise e = new Entreprise(); // création d'une entreprise
	e.setAdresse("1");
	e.setTelephone("0141220300");
	e.setNomSocial("SQLi");
	e.setSiret("44019981800012");
	EntityManager em = emf.createEntityManager();
	EntityTransaction transaction = em.getTransaction();
	transaction.begin();
	em.persist(e);

		Particulier p = new Particulier(); // création d'un particulier
	p.setAdresse("14 rue de la Foret");
	p.setTelephone("0112341234");
	p.setNom("Durant");
	p.setPrenom("Cédric");
	em.persist(p);

		transaction.commit();
	Entreprise e2 = (Entreprise) em.find(AbstractClient.class, 3);
	Particulier p2 = (Particulier) em.find(AbstractClient.class, 4);
	LOGGER.debug("Client: " + p2);
	LOGGER.debug("Client: " + e2);
}
```

1 table par classe concrète

- Appellation: « Table per class »

- Idée générale
  - Chaque classe concrète est stockée dans une table différente
  - Duplication des colonnes dans le schéma pour les propriétés communes

- Théoriquement
  - (-) Équivaut à dire: « ne considérons pas la relation d'héritage, ce sont des classes différentes sans lien particulier entre elles »

- Techniquement
  - (-) Pas d'unicité globale des clés primaires sur toutes les tables
  - (+) Pas de problème de colonnes vides
  - (+) Pas de clés étrangères → pas de problème de jointure

1 table par classe concrète

- Client est abstraite → Pas de table
- Particulier est concrète → 1 table PARTICULIER
  - des colonnes sont ajoutées pour les attributs de Client
- Entreprise est concrète → 1 table ENTREPRISE
  - des colonnes sont ajoutées pour les attributs de Client

ID	ADRESSE	PHONE	NOM	PRENOM
234	13 rue des …	014565…	Martin	Patrice
887	147 avenue …	042356…	Durant	Cédric

ID	ADRESSE	PHONE	SIRET	NSOC
14	15 place du …	032981…	12462…	S&B SA
516	3 impasse …	029867…	98765…	InterFilm

clés primaires

1 table par classe concrète

```java
/**
 * Client abstrait: Entreprise ou particulier.
 */
@MappedSuperclass
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class AbstractClient {

    @Id
    private Integer id;
}
```

L'entité mère n'est pas mappée à une table. Les requêtes polymorphes sont rendus impossibles par cette annotation

```java
/**
 * Modélise une entreprise.
 */
@Entity
public class Entreprise extends AbstractClient {

    private String nomSocial;
    private String siret;
}
```

```java
/**
 *Un particulier.
 */
@Entity
public class Particulier extends AbstractClient {

        private String nom;
        private String prenom;
}
```

1 table par classe concrète

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasseConcrete() {
    Entreprise e = new Entreprise(); // création d'une entreprise
    e.setId(3);
    e.setAdresse("1");
    e.setTelephone("0141220300");
    e.setNomSocial("SQLi");
    e.setSiret("44019981800012");
    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();
    em.persist(e);

    Particulier p = new Particulier(); // création d'un particulier
    e.setId(3);
    p.setAdresse("14 rue de la Foret");
    p.setTelephone("0112341234");
    p.setNom("Durant");
    p.setPrenom("Cédric");
    em.persist(p);

    transaction.commit();
    Entreprise e2 = em.find(Entreprise.class, 3);
    Particulier p2 = em.find(Particulier.class, 4);
    LOGGER.debug("Client: " + p2);
    LOGGER.debug("Client: " + e2);
}
```

Chapitre 6
Clé composite

Exemple de clé composite

```
@Entity
public class Project {
        @EmbeddedId
        private CleComposite id;
}

@Embeddable
class CleComposite {
    private int departmentId;
    private long projectId;
}
```

Chapitre 7
JPQL

Sommaire

JPQL: Pourquoi ?

Paramètre nommé ou positionnel

Les divers types de retour

Opérateur new

Clause JOIN

Update et DELETE

Utilisation des fonctions size et is (not) empty

JPQL: pourquoi ?

- La méthode find est vite limitée (recherche par id uniquement).

- JPQL, language standardisée indépendant de la base de données
  - Proche du SQL
  - S'utilise avec les **entités JPA** et leurs **attributs**.

```
SELECT user FROM User user WHERE user.login LIKE :LOGIN
```

```
SELECT count(user) FROM User user
```

Paramètre nommé ou positionnel

- Paramètre nommé

```
Query query = em.createQuery("SELECT p FROM Person p WHERE p.name= :name");
query.setParameter("name", "Doe");
```

- Paramètre positionnel

```
Query query = em.createQuery("SELECT p FROM Person p WHERE p.name= ?");
query.setParameter(1, "Doe");
```

Les divers types de retour (1/2)

- Liste typée

```
TypedQuery<Person> query = em.createQuery("SELECT p FROM Person p WHERE p.age > 50", Person.class);
List<Person> persons = query.getResultList();
```

- Résultat unique

```
TypedQuery<Person> query = em.createQuery("SELECT p FROM Person p WHERE p.id = ?", Person.class);
query.setParameter(1, 12);
Person person = query.getSingleResult();
```

Les divers types de retour (2/2)

Valeur unique

```
Query query = em.createQuery("SELECT MAX(p.age) FROM Person p");
BigDecimal ageMax = (BigDecimal)query.getSingleResult();
```

Ensemble de valeurs

```
Query query = em.createQuery("SELECT p.firstName FROM Person p");
List<String> firstNames = query. getResultList();
```

Liste d'ensembles de valeurs

```
Query query = em.createQuery("SELECT p.firstName, p.lastName FROM Person p");
List<Object[]> firstNamesAndLastNames = query.getResultList();
```

Opérateur new

- Permet d'instancier un objet dans une requête JPQL

```
Query query = em.createQuery("SELECT new fr.person.Customer(p.firstName, p.lastName) FROM Person p");
List<Customer> customers = query.getResultList();
```

- Remarques
  - La classe Customer **n'est pas forcément** une entité.
  - Attention en cas de refactoring.

Clause JOIN

- Pour réaliser une jointure

- Exemple 1

```
Query query = em.createQuery("SELECT p FROM Person p JOIN p.address a WHERE a.city= 'Nantes'");
List<Person> persons = query.getResultList();
```

- Exemple 2

```
Query query = em.createQuery("SELECT l FROM Livre l JOIN l.emprunts e WHERE e.delai > 5");
List<Livre> livres = query.getResultList();
```

UPDATE et DELETE

- Pour réaliser un update

```
Query query = em.createQuery("UPDATE Livre l SET l.titre= :newTitre WHERE l.titre=:oldTitre");
query.setParameter("newTitre", "Germinal");
query.setParameter("oldTitre", "Germinol");
query.executeUpdate();
```

- Pour réaliser un delete

```
Query query = em.createQuery("DELETE FROM Livre l WHERE l.titre=:titre");
query.setParameter("titre", "Germinol");
query.executeUpdate();
```

FONCTIONS

❑ Fonction SIZE

```
Query query = em.createQuery("SELECT l FROM Livre l WHERE SIZE(l.emprunts)=0");
List<Livre> livres = query.getResultList();
```

❑ Fonction IS (NOT) EMPTY

```
Query query = em.createQuery("SELECT l FROM Livre l WHERE l.emprunts IS EMPTY");
List<Livre> livres = query.getResultList();
```

Atelier (TP)

OBJECTIFS : Utiliser JPA

DESCRIPTION :

- Dans les TP n°4 à 6 vous allez mettre en place des annotations JPA dans divers contextes

Annexe
Cache L2

Sommaire

JPA et cache L2

- Par défaut JPA fonctionne avec un cache de niveau 1 associé à l'EntityManager.

- JPA peut fonctionner avec un **cache de niveau 2**, associé à l'EntityManagerFactory, et donc actif durant toute la durée de vie de l'application.

- On va stocker dans ce cache prioritairement des informations de type "référentiel", i.e. qui changent peu.

- Exemples de données "référentiel":
  - *Une liste de salles de formation pour une application de gestion d'une société de formation.*
  - *La liste des départements français*
  - *La liste des pays du monde*
  - *etc.*

Cache L2 - dépendances

- La première étape est d'ajouter une dépendance dans le fichier pom.xml.

- Ici on choisit l'implémentation ehcache:

```
<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-ehcache</artifactId>
 <version>…</version>
</dependency>
```

Cache L2 - configuration

- La seconde étape est d'activer le cache dans le fichier persistence.xml.

- Ici on choisit l'implémentation ehcache:

```
...
<property name="hibernate.cache.use_second_level_cache" value="true" />
<property name="javax.persistence.sharedCache.mode" value="ENABLE_SELECTIVE"/>
<property name="hibernate.cache.region.factory_class" value="org.hibernate.cache.ehcache.EhCacheRegionFactory" />
...
```

Seules les entités marquées de l'annotation **@Cacheable** sont prises en compte.

- Autre mode possible pour sharedCache.mode:
  - NONE — aucune entité en cache
  - ALL — toutes les entités en cache
  - DISABLE_SELECTIVE — toutes les entités sauf celles avec @Cacheable(false)
  - UNSPECIFIED — mode par défaut.

Cache L2 – annotation des entités

La dernière étape est d'annoter les entités avec une annotation particulière.

```
@Entity
@Table(name = "LIVRE")
@Cacheable
public class Livre {
 …
}
```

Cache L2 – fichier ehcache.xml

- Il est possible de configurer le cache plus finement avec un fichier de configuration à placer dans src/main/resources.

- Dans ce cas une nouvelle propriété doit être ajoutée dans persistence.xml:

```
<property name="hibernate.cache.provider_configuration_file_resource_path" value="ehcache.xml"/>
```

- Exemple:

```
<ehcache xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance xsi:noNamespaceSchemaLocation="ehcache.xsd"
        updateCheck="true" monitoring="autodetect" dynamicConfig="true">

        <defaultCache name="defaut" maxElementsInMemory="5000" eternal="true"
                    overflowToDisk="false" diskPersistent="false"
                    memoryStoreEvictionPolicy="LRU"/>
</ehcache>
```

Cache L2 – vérification programmatique

- Il est possible de vérifier si un objet est dans le cache ou non.

```
Cache cache = entityManagerFactory.getCache();
boolean isInCache = cache.contains(Livre.class, 1);
```