Java Persistence API 2.0 – JSR 317

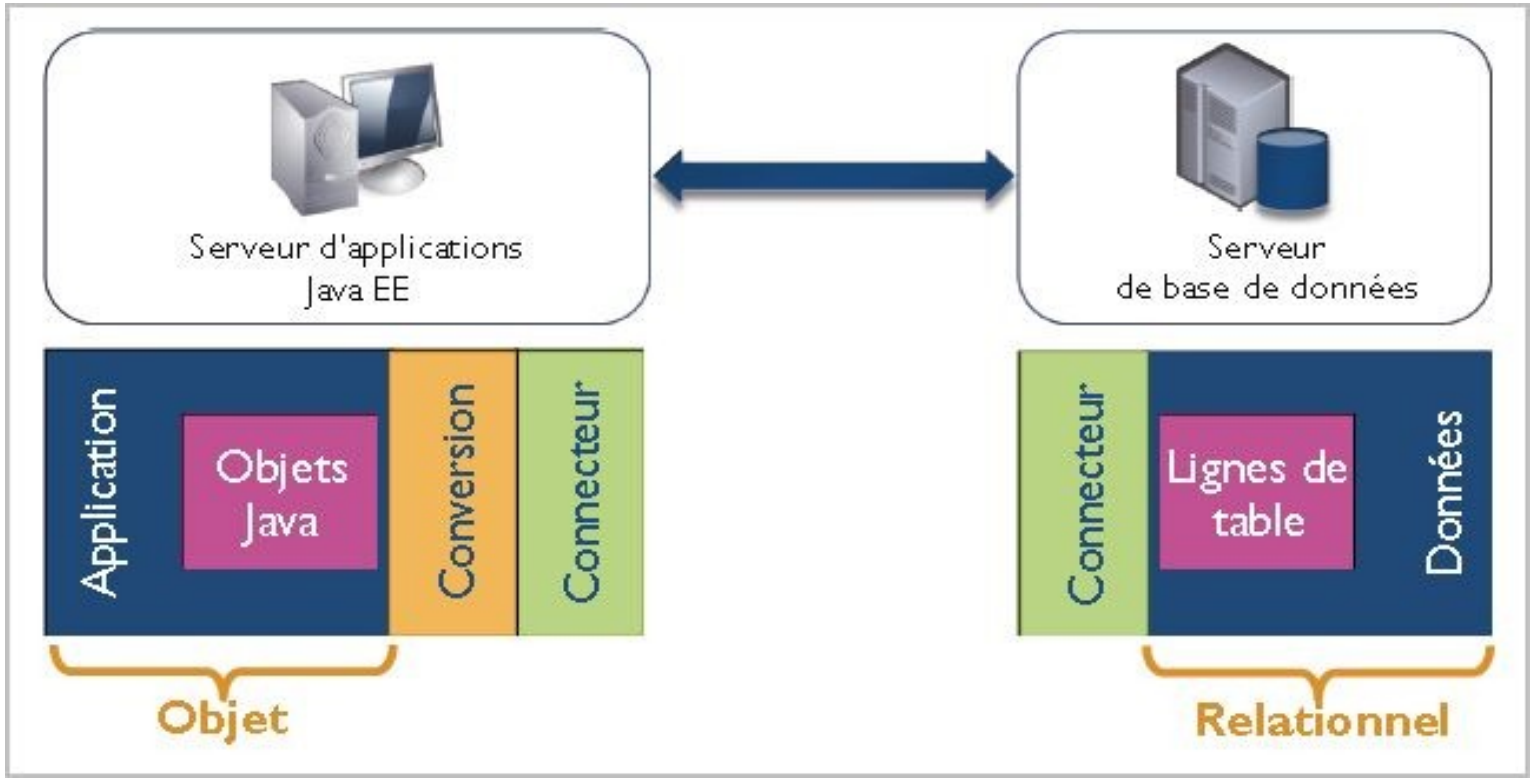
Audran YOULOU ZHAZHA, JCertif University 2012
<audran.zhazha@gmail.com>

Plan

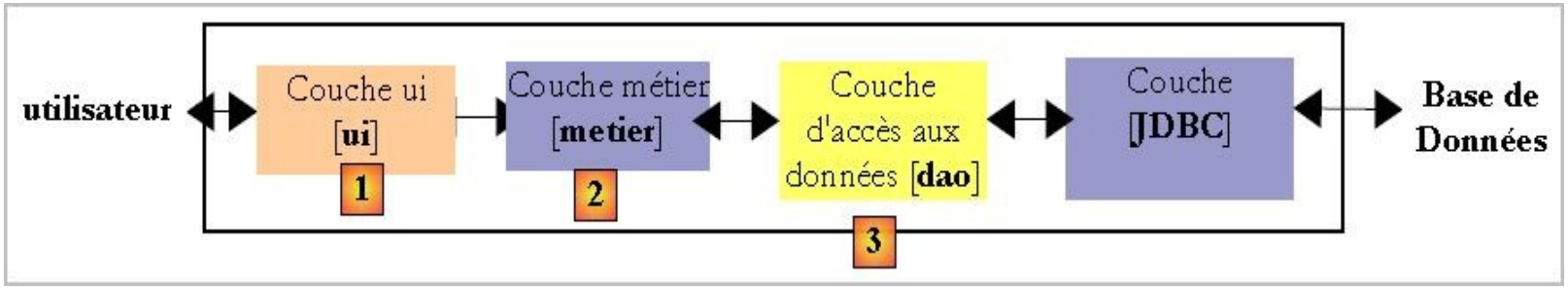
- Introduction
- Qu’est ce que JPA ?
- Entités et gestionnaire d’entités
- Métadonnées
- Mapping des entités et héritage
- Gestion des objets persistants
- Callbacks & Listeners
- Questions

Introduction

- Persistance en base de données
  - **Besoin**: Create + Read + Update + Delete (CRUD)
  - **Problème**: mapping objet / relationnel des données ?



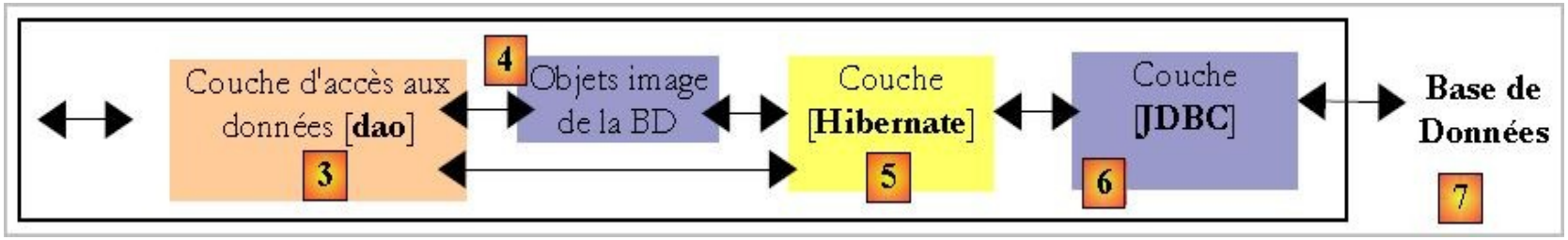
Introduction



- Couche[1] : U.I (swing, console ou browser)
- Couche[2] : implante la logique métier
- Couche[3] : couche [dao], **fournit des données** à la couche métier
- Couche[JDBC] : standard en java pour **accéder à une B.D**. C'est le pilote JDBC

Introduction (suite)

- JPA vise **à faciliter l'écriture** de la couche[dao], celle qui gère les données dites persistantes, d'où le nom de l'API : **Java Persistence API**
- Hibernate, une des solutions ORM qui a révolutionné la persistance avec java



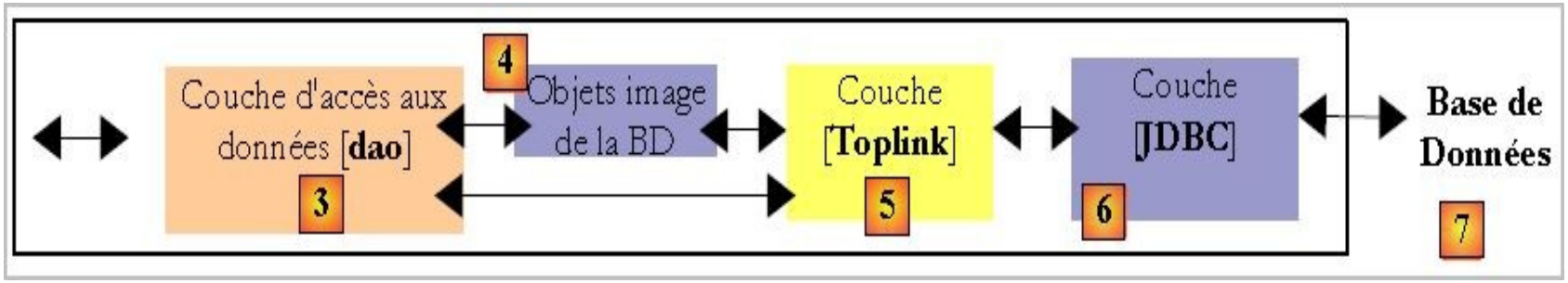
- La couche[Hibernate] vient se placer entre la couche[dao] écrite par la développeur et la couche[JDBC]

Introduction (suite)

- Le développeur de la couche [dao] ne voit plus la couche [JDBC]
- La couche [Hibernate] est une couche d'abstraction qui se veut la plus transparente possible
- **L'idéal visé est que le développeur de la couche [dao] puisse ignorer totalement qu'il travaille avec une base de données**
- La couche [4] des objets, image de la B.D est appelée "**contexte de persistance**"

Introduction (suite)

Si, l'architecture précédente devient la suivante :

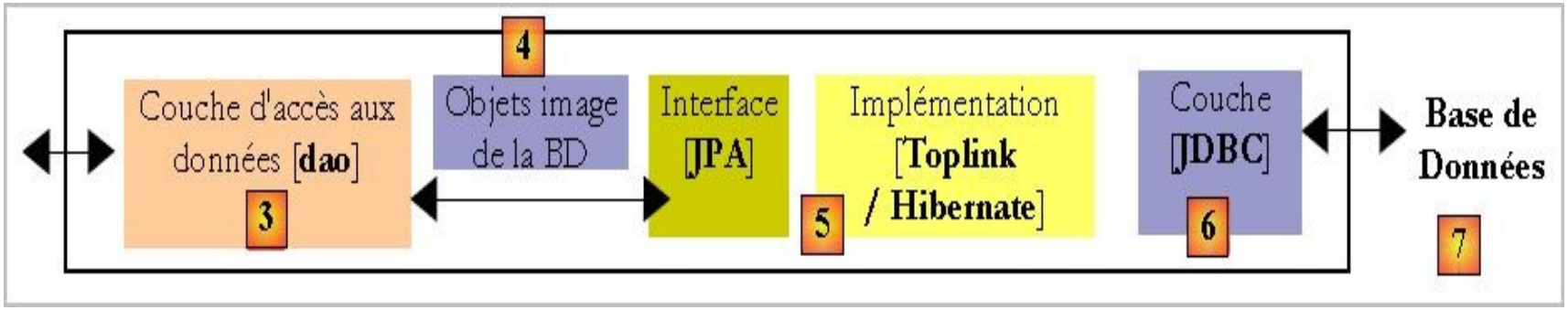


- Le développeur **doit également changer** sa couche[dao] qui avait été écrite pour dialoguer avec un ORM spécifique (Hibernate)

- Devant le succès des ORM et du manque de spécifications Java sur les ORM, **Sun décida alors de créer une spécification JPA.**

Introduction (suite)

Avec JPA, l'architecture précédente devient la suivante :



- La couche [dao] dialogue maintenant avec la spécification JPA, **un ensemble d'interfaces**

- Quelque soit le produit qui implémente celle-ci, l'interface de la couche JPA présentée à la couche [dao] reste la même.

Qu’est ce que JPA ?

- Standard de persistance dans un RDBMS

  - JPA 1.0 a été introduite dans JEE5

  - JPA 2.0 est une partie des standards JEE6

- JPA apporte des simplifications significatives

  - ORM (Object / Relational mapping)

  - Utilisation des annotations

  - Les entités sont des POJOs

  - l’API JPA peut être utilisée en dehors du conteneur EJB

Qu’est ce que JPA ?

- Implémentations de JPA

  - EclipseLink 2.0 (Impl. de reférence)

  - OpenJPA (Apache)

  - TopLink (WebGain)

  - Hibernate (JBoss)

  …

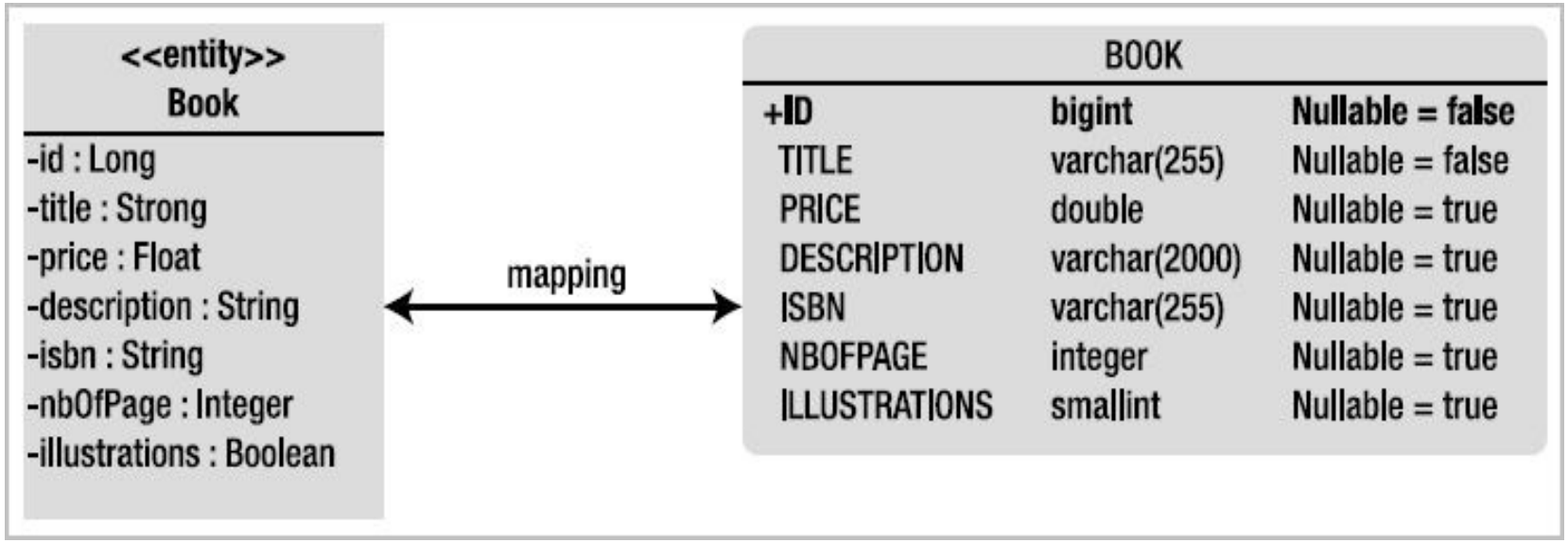
Entités et gestionnaire d'entités

- Entité: c'est une **classe persistante** ou une **instance** d'une classe persistante.
- Exemple d'entité: Book

```
@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

Entités et gestionnaire d'entités (suite)

- L'entité **Book** est mappée à la table **BOOK**

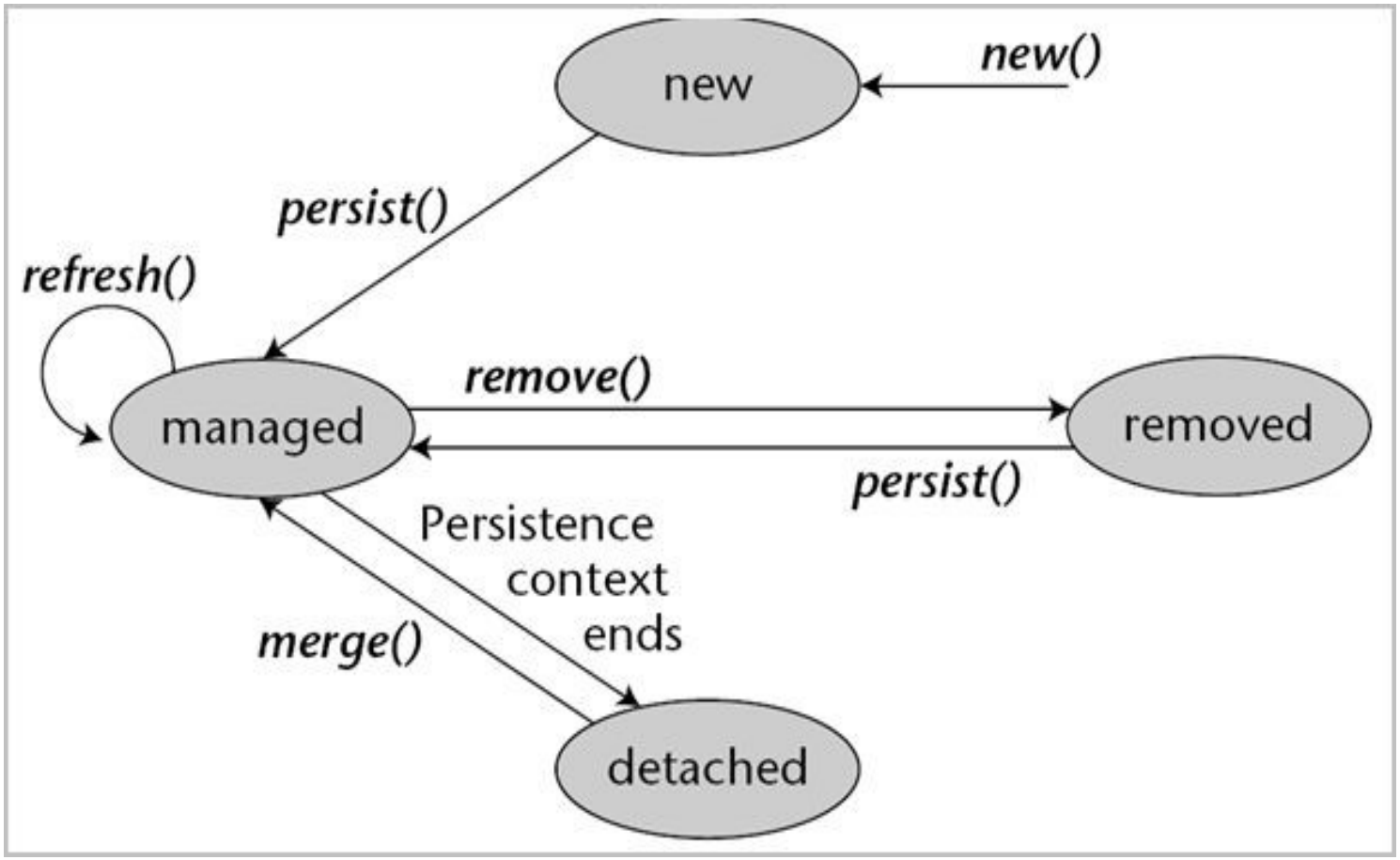


- Un objet qui n'est pas persistant est dit « **transient** », c'est un objet temporaire

Entités et gestionnaire d'entités (suite)

- EntityManager: c'est la pièce centrale de l'API JPA qui est responsable de l'orchestration des entités
- L'EntityManager est une interface chargée de la gestion des entités, fournie des opérations CRUD sur les entités et assure la synchronisation du contexte de persistance avec la B.D

Entités et gestionnaire d'entités (suite)



Cycle de vie d'un objet manipulé avec le gestionnaire d'entités

Métadonnées

- Dans JPA, les annotations sont exploitées pour définir la mise en correspondance des classes persistantes avec le modèle relationnel et sont appelées **métadonnées**.
- Le fournisseur de persistance accède à l'état de l'entité soit par ses **variables d'instance**, soit par ses **accesseurs**
- Si la variable d'instance est annotée: elle sera donc utilisée directement (**field based access**)
- Si le getter est annoté: les accesseurs seront utilisés (**property based access**)

Métadonnées

- Les annotations puisent le cœur de l'information depuis l'élément sur lequel elles s'appliquent
- Contrairement aux anciens descripteurs de déploiement des EJB entité, elles sont standardisées et donc portables
- Elles sont, à de rares exceptions, beaucoup moins verbeuses que le XML
- Elles bénéficient d'une phase de compilation qui permet de valider en direct qu'elles sont, au minimum, syntaxiquement correctes

Mapping des entités

- La classe de l'entité doit être annotée avec **@javax.persistence.Entity**
- L'annotation **@javax.persistence.Id** doit être utilisée pour une clé primaire simple
- La classe de l'entité doit avoir un constructeur sans argument
- Une « Enum » ou une « interface » ne peut être désignée comme entité
- L'entité ne doit pas être « final ».

Mapping des entités

- Si une instance de l'entité peut être passée par valeur comme objet détaché du contexte de persistance, la classe de l'entité doit implémenter l'interface **Serializable**
- Par défaut, le **nom de l'entité** est mappée à la **table** correspondante
- Par défaut, les **attributs** sont mappés aux **colonnes** de la table correspondantes
- Les types primitives java (**String, Long, Boolean**,…) sont mappés à (**VARCHAR**, **BIGINT**, **SMALLINT**,…)

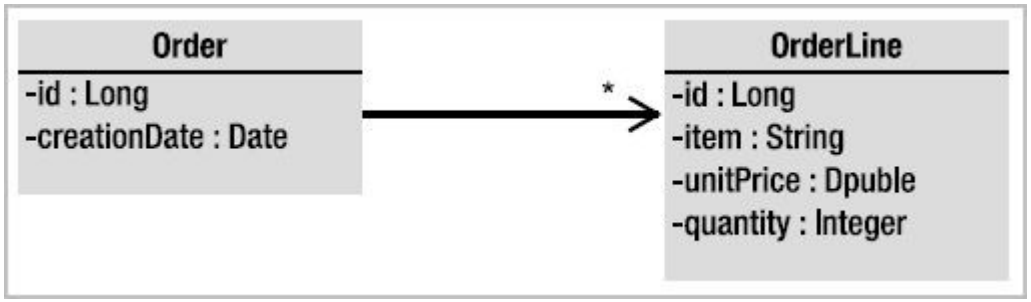
Mapping des entités – association « OneToOne »

```java
@Entity
public class Customer

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```

CUSTOMER		
+ID	bigint	Nullable = false
LASTNAME	varchar(255)	Nullable = true
PHONENUMBER	varchar(255)	Nullable = true
EMAIL	varchar(255)	Nullable = true
FIRSTNAME	varchar(255)	Nullable = true
#ADDRESS_FK	*bigint*	*Nullable = true*

ADDRESS		
+ID	bigint	Nullable = false
STREET2	varchar(255)	Nullable = true
STREET1	varchar(255)	Nullable = true
ZIPCODE	varchar(255)	Nullable = true
STATE	varchar(255)	Nullable = true
COUNTRY	varchar(255)	Nullable = true
CITY	varchar(255)	Nullable = true

Mapping des entités – association « OneToMany »



Une commande (Order) est associée à plusieurs lignes de commandes (OrderLine)

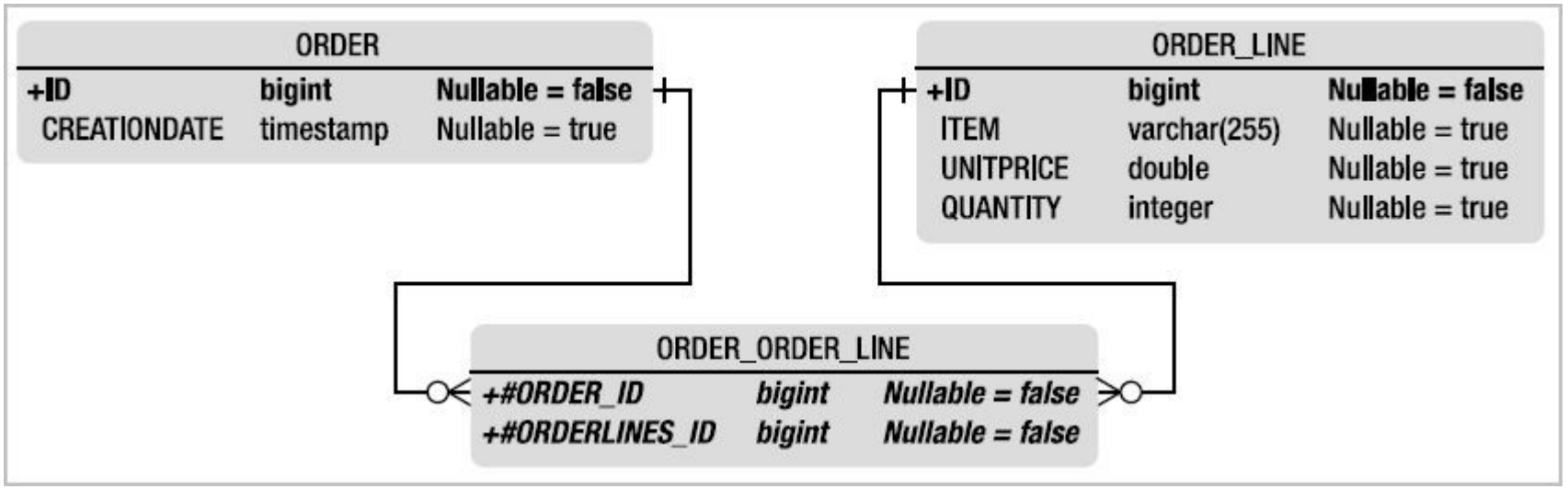
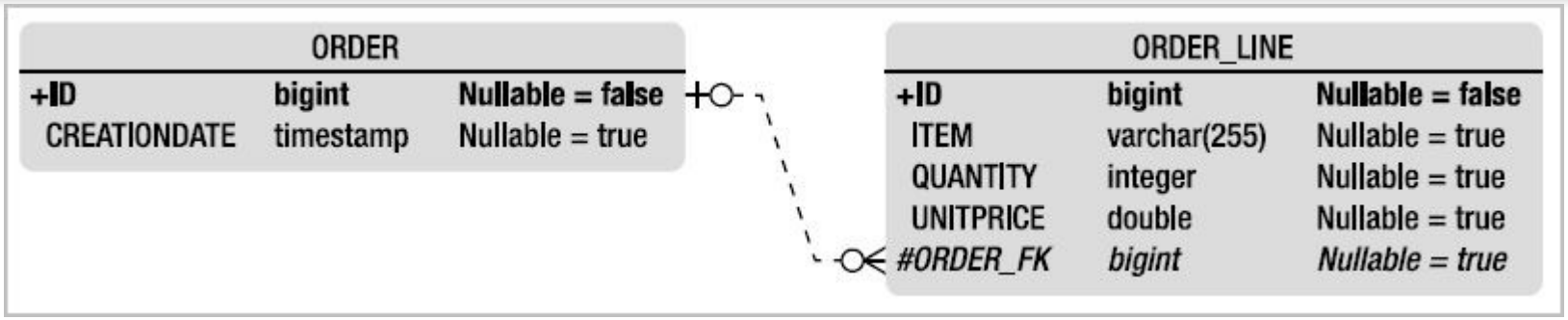


Table de jointure entre les tables ORDER et ORDER_LINE

Mapping des entités – association « OneToMany » via table de jointure

```
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany
    @JoinTable(name = "jnd_ord_line",
        joinColumns = @JoinColumn(name = "order_fk"),
        inverseJoinColumns = @JoinColumn(name = "order_line_fk") )
    private List<OrderLine> orderLines;

    // Constructors, getters, setters
}
```

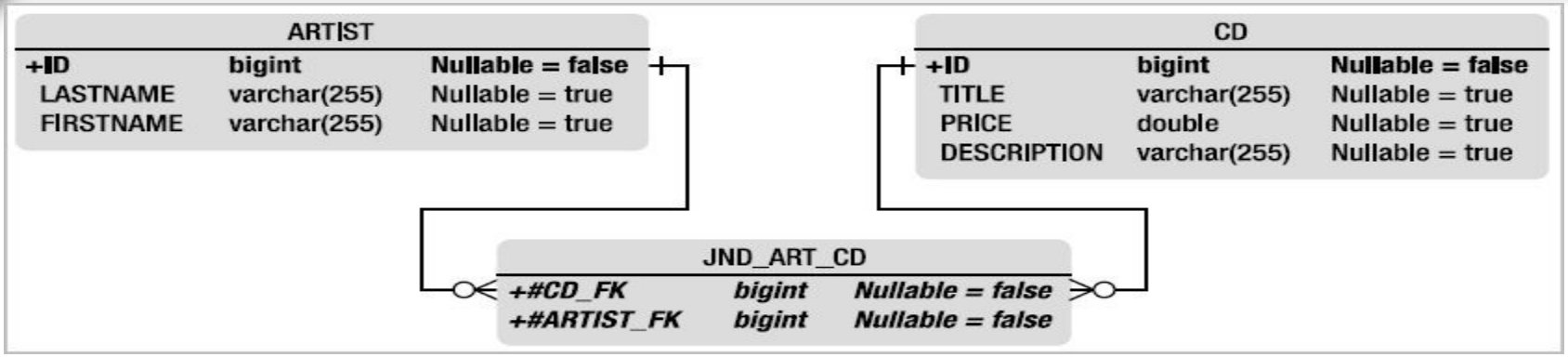
Mapping des entités – association « OneToMany » via colonne de jointure



Association entres les tables ORDER et ORDER_LINE

```
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany(fetch = FetchType.EAGER)
    @JoinColumn(name = "order_fk")
    private List<OrderLine> orderLines;

    // Constructors, getters, setters
}
```

Mapping des entités – association « ManyToMany »



Association entre les tables ARTIST et CD, via la table JND_ART_CD

```
@Entity
public class Artist {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @ManyToMany
    @JoinTable(name = "jnd_art_cd", ↪
        joinColumns = @JoinColumn(name = "artist_fk"), ↪
        inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD> appearsOnCDs;

    // Constructors, getters, setters
}
```

```
@Entity
public class CD {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    @ManyToMany(mappedBy = "appearsOnCDs")
    private List<Artist> createdByArtists;

    // Constructors, getters, setters
}
```

Mapping des entités et héritage

- Une table par classe concrète (**TABLE_PER_CLASS**). Optionnelle.
- Une table par sous classe (**JOINED**). Il s'agit d'utiliser une table par sous classe en plus de la table mappée à la classe mère. Peu performante, et garantit les contraintes d'intégrité.
- Une table par hiérarchie de classes (**SINGLE_TABLE**). Excellente pour les performances et autorise le polymorphisme. Ne garantit pas les contraintes d'intégrité comme les clauses « NOT NULL ».

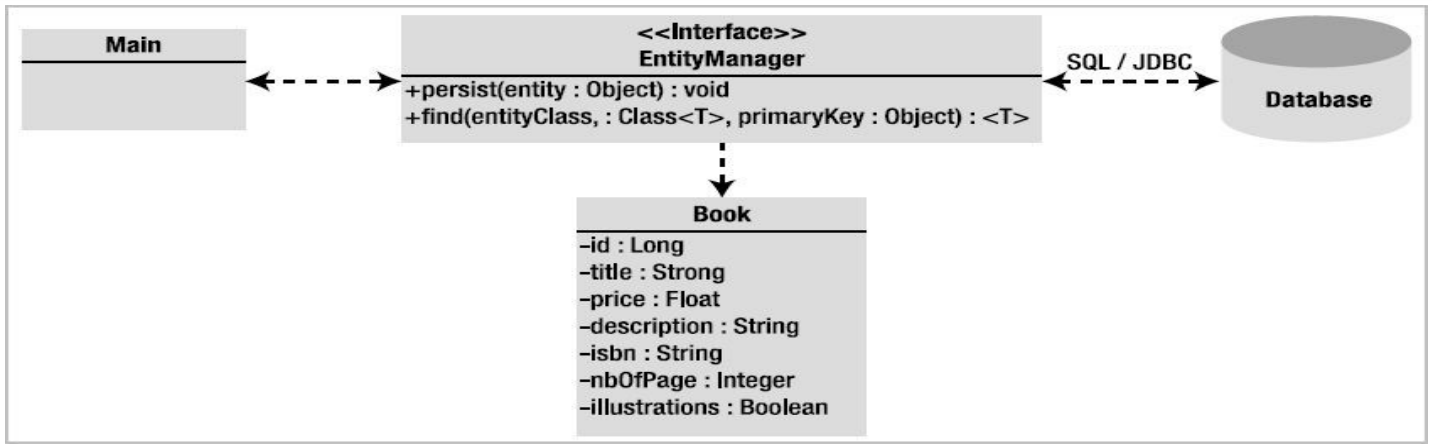
Gestion des objets persistants – utilisation de l'interface EntityManager (JSE)

```
public class Main {
        public static void main(String[] args) {
                // 1-Create an instance of the Book entity
                Book book = new Book();
                book.setId(1234L);
                book.setTitle("Introduction à JPA");
                book.setPrice(12.5F);
                book.setDescription("Spec. created by JCP");
                book.setIsbn("1-84023-742-2");
                book.setNbOfPage(354);
                book.setIllustrations(false);

                // 2-Get an entity manager and a transaction
                EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("jpaPU");
                EntityManager em = emf.createEntityManager();
                EntityTransaction tx = em.getTransaction();

                // 3-Persist the book to the database
                tx.begin();
                em.persist(book);
                tx.commit();

                // 4-Retrieve the book by its identifier
                book = em.find(Book.class, 1234L);
                System.out.println(book);

                em.close();
                emf.close();
        }
}
```

Gestion des objets persistants – utilisation de l'interface EntityManager (JEE)

```
@Stateless
public class BookBean {
        @PersistenceContext(unitName = "jpaPU")
        private EntityManager em;

        public void createBook() {
                // Create an instance of book
                Book book = new Book();
                book.setId(1234L);
                book.setTitle("Introduction à JPA");
                book.setPrice(12.5F);
                book.setDescription("Spec. created by JCP");
                book.setIsbn("1-84023-742-2");
                book.setNbOfPage(354);
                book.setIllustrations(false);

                // Persist the book to the database
                em.persist(book);

                // Retrieve the book by its identifier
                book = em.find(Book.class, 1234L);
                System.out.println(book);
        }
}
```

Gestion des objets persistants – utilisation de l'interface EntityManager



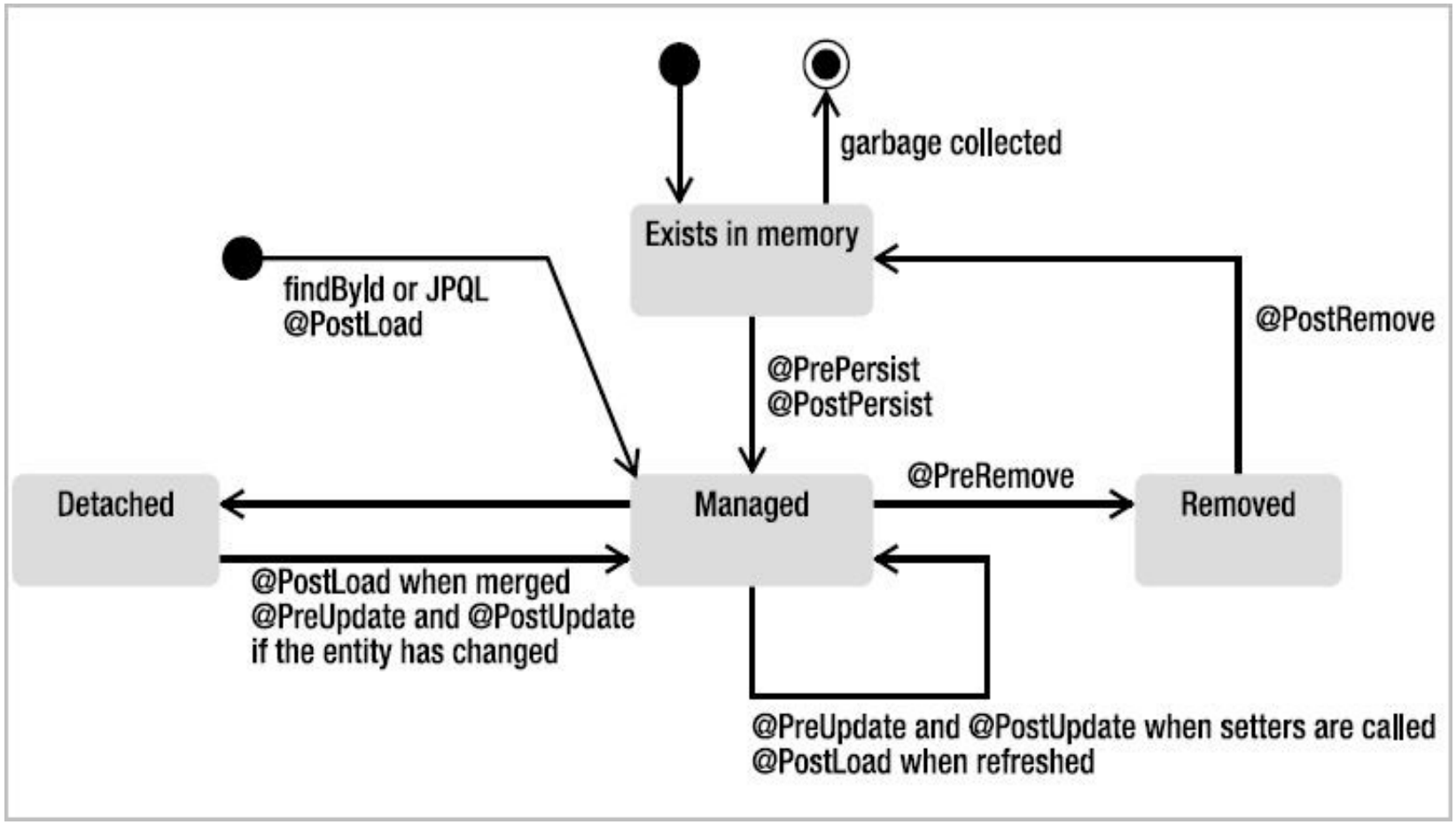
Interaction du gestionnaire d'entités avec l'entité et la B.D

- Quel JDBC driver utilisé ?
- Comment se connecter à la B.D ?
- Quel est le nom de la base de données ?
- Ces infos. sont dans le fichier **persistence.xml**

Gestion des objets persistants – unité de persistance

```
<?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="jpaPU" transaction-type="RESOURCE_LOCAL">
                <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
                <class>com.jcertif.university.lab.model.Book</class>
                <properties>
                        <property name="eclipselink.target-database" value="DERBY"/>
                        <property name="javax.persistence.jdbc.driver"
                                value="org.apache.derby.jdbc.ClientDriver"/>
                        <property name="javax.persistence.jdbc.url"
                                value="jdbc:derby://localhost:1527/CDStoreDB"/>
                        <property name="javax.persistence.jdbc.user" value="root"/>
                        <property name="javax.persistence.jdbc.password" value="root"/>
                </properties>
        </persistence-unit>
</persistence>
```

Callbacks – annotations



Cycle de vie d'une entité avec les annotations de callbacks

Callbacks – annotations

- **@PrePersist**: invoquée avant que la méthode **EntityManager.persist()** soit exécutée
- **@PostPersist**: invoquée après la persistance de l'entité
- **@PostLoad**: invoquée après la chargement de l'entité avec une requête JPQL ou via **EntityManager.find()**
- **@PreRemove**: invoquée avant que la méthode **EntityManager.remove()** soit exécutée

Callbacks – méthodes et prérequis

- Elles peuvent être avoir les accès « public, private, protected ou package ». Mais, elles ne peuvent pas être « static ou final »
- Elles peuvent avoir plusieurs annotations callbacks
- Elles peuvent lancer les exceptions non contrôlées (RuntimeException), mais pas les exceptions contrôlées
- Elles peuvent invoquer JNDI, JDBC, JMS et EJB mais ne peuvent pas invoquer les méthodes de **EntityManager** ou **Query**

Callbacks - exemple

```
@Entity
public class Customer {
	// Fields

	@PrePersist
	@PreUpdate
	private void validate() {
		if (firstName == null || "".equals(firstName))
			throw new IllegalArgumentException("Invalid first name");
		if (lastName == null || "".equals(lastName))
			throw new IllegalArgumentException("Invalid last name");
	}

	@PostLoad
	@PostPersist
	@PostUpdate
	public void calculateAge() {
		if (dateOfBirth == null) {
			age = null;
			return;
		}

		Calendar birth = new GregorianCalendar();
		birth.setTime(dateOfBirth);
		Calendar now = new GregorianCalendar();
		now.setTime(new Date());

		int adjust = 0;
		if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
			adjust = -1;
		}

		age = now.get(YEAR) - birth.get(YEAR) + adjust;
	}

	// Constructors, getters, setters
}
```

Listeners

- Un listener permet d'extraire la logique métier d'une entité
- Un listener implémente la logique métier dans une classe dédiée
- Un listener permet aux entités de partager la même logique métier
- Un listener est un POJO, dans lequel on peut définir une ou plusieurs annotations callbacks
- Pour utiliser un listener, une entité doit utiliser l'annotation **@EntityListeners**

Listeners - exemples

```
//A Listener Calculating the Customer's Age
public class AgeCalculationListener {
        @PostLoad
        @PostPersist
        @PostUpdate
        public void calculateAge(Customer customer) {}
}

//A Listener Validating the Customer's Attributes
public class DataValidationListener {
        @PrePersist
        @PreUpdate
        private void validate(Customer customer) {}
}

//The Customer Entity Defining Two Listeners
@EntityListeners({DataValidationListener.class,
AgeCalculationListener.class})
@Entity
public class Customer {
        // Attributes

        // Constructors, getters, setters
}
```

Questions ?