Java Persistence API (JPA)

Plan

- 1- ORM (Object Relational Mapping)
- 2- Qu'est ce que le JPA ?
- 3- Exemple de MaPPING Hibernate
- 4- Les associations (mapping)

ORM (Object Relational Mapping)

- **l'ORM** permet donc à transformer une table en un objet facilement manipulable via ses attributs qui correspondent aux champs de cette table, manipuler et accéder à ces objets sans considérer comment ces objets sont liés à leur source de données.

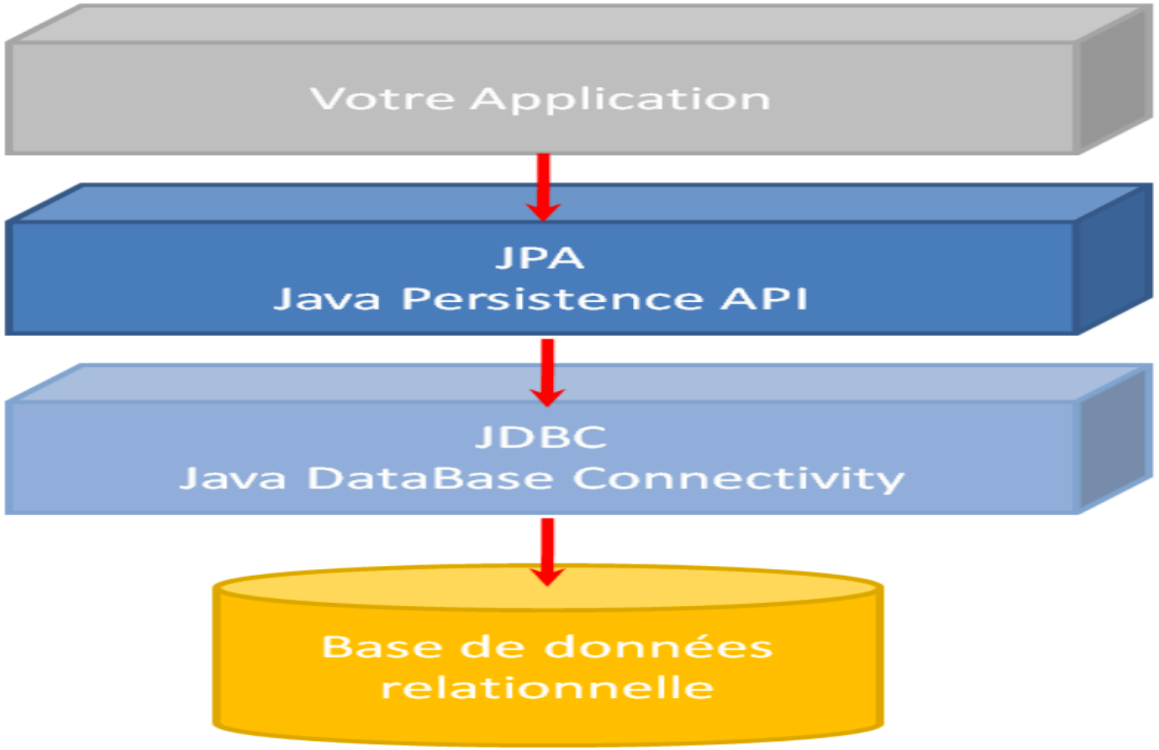
  la JDBC (Java DataBase Connectivity) a apporté une standardisation au niveau de l'accès des bases de données, mais ça nécessite l'écriture du code pour réaliser le CRUD (CREATE, READ, UPDATE, DELETE) et réaliser le mapping entre tables et objets.

- Par contre **l'ORM** permet la réutilisabilité du code et la réduction de sa quantité, afin d'effectuer les opérations de base (CRUD) et on pense plutôt en termes d'objet et pas en termes de lignes de tables.

  Cette technique de programmation est largement répandue dans le monde Java.

La persistance des données avec JPA

- On peut considérer JDBC comme une API de bas niveau. La plate-forme JEE propose une couche d'abstraction par-dessus JDBC : JPA (Java Persistence API).



- JPA ne permet d'adresser qu'à des bases de données relationnelles (à l'instar de JDBC).

Qu'est ce que le JPA ?

- **JPA** est une spécification d'ORM pour Java.
  il en existe plusieurs implémentations de **JPA**.
- Les trois plus implémentations (Framework) connus sont:
  - **Hibernate** (RedHat),
  - **Eclipse Link** (de la fondation Eclipse, vous vous en doutez)
  - **Open JPA** (de la fondation Apache).

  Remarque: Le serveur Glassfish embarque l'implémentation Eclipse Link.
- Les intérêts à utiliser un ORM compatible JPA sont:
  - **Abstraction quasi totale du langage SQL**
  - **Indépendance vis-à-vis de la base de données** Vous êtes donc libre de changer votre SGBDr (Système de Gestion de Base de Données relationnel) à tout moment.
  - **Meilleure productivité** : comparer à JDBC, vous avez beaucoup moins de code à produire (et notamment avec le SQL).
  - **Risque d'injection SQL fortement réduit** : vu que vous ne produisez plus de codes SQL, vous évitez un grand nombre d'attaques basées sur une mauvaise utilisation de ce langage.

Un premier Exemple d'utilisation JPA

- Dans ce premier exemple **JPA**, nous allons chercher à mapper une table, en base de données, contenant des articles à une classe Article.

- Nous allons juste lancer un programme Java, utilisant JPA, en mode console(nous n'aurons pas accès par défaut, à une implémentation JP Il faut donc en télécharger une implémentation **JPA** (Ex. Hibernate)

- CREATE TABLE **T_Personne** (

IdArticle	int	PRIMARY KEY AUTO_INCREMENT,
Nom	text	NOT NULL,
Prenom	text	NOT NULL);

La classe Personne

- Il nous faut maintenant produire une classe Java permettant la manipulation des personnes

```
package com.hibernate;

public class Personne {
    private int idPersonne;
    private String nom;
    private String personne;

      public Personne() {
        this( "unknown", "unknown");
    }
    public Personne(String nom, String personne) {
        this.nom = nom;
        this.personne = personne;
    }
    public int getIdPersonne() {
        return idPersonne;
    }
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPersonne() {
        return personne;
    }
  public void setPersonne(String personne) {
        this.personne = personne;
    }
@Override
public String toString() {
    return "Personne [idPersonne=" + idPersonne + ", nom=" + nom + ", per
}
```

Le Mapping JPA

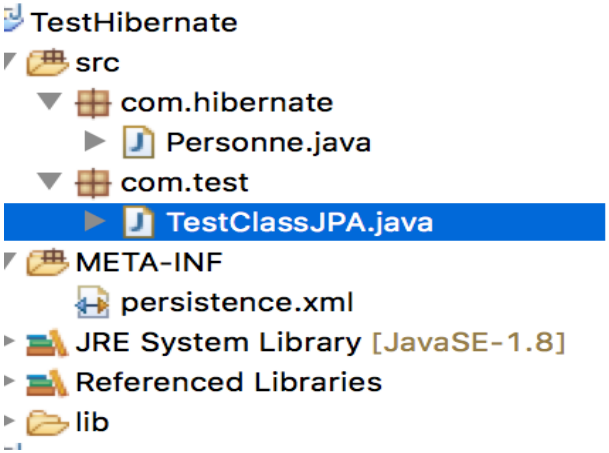
- Nous allons maintenant traiter de la partie la plus importante : le mapping JPA. C'est lui qui va faire le lien entre le monde de la base de données et le monde Objet. Ce mapping se définit sur votre classe grâce à un jeu d'annotations.

```
import javax.persistence.*;

@Entity
@Table( name="T_Personne" )
public class Personne {
    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private int idPersonne;

    private String nom;

    @Column( name="Pprenom" )
    private String prenom;

      public Personne() {
        this( "unknown", "unknown");
    }
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
```

Le Mapping JPA

- L'annotation **@Entity** precise que la classe Personne est soumisse à persistance : c'est une entité persistante.
- L'annotation **@Table** permet de spécifier le nom de la table (en base de données) dans laquelle nos instances de personnes devront être persistées.
  Rq: Par défaut, la table portera le même nom que la classe.
- L'annotation **@Id** indique que l'attribut idPersonne de votre classe est « mappé » (mis en correspondance, si vous préférez) à la colonne de clé primaire en base de données
- L'annotation **@GeneratedValue** indique que le moteur de base de données à la responsabilité de générer les nouvelles valeurs pour la clé primaire en cas d'insertion de nouveaux objets.
- Par défaut, un attribut est mappé à une colonne de même nom en base de données. Si vous ne souhaitez pas mapper un attribut, indiquez-le via l'annotation **@Transient**.
- L'annotation **@Column** utilisée pour le prénom des personnes permet d'indiquer le nom de la colonne correspondante en base de données, étant donné que les deux noms ne correspondent pas.

La configuration de la persistance




```
    <persistence-unit name="TestJpa">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <class>com.hibernate.Personne</class>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/TestJpa" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value=" " />

            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
        </properties>
    </persistence-unit>

</persistence>
```

La configuration de la persistance

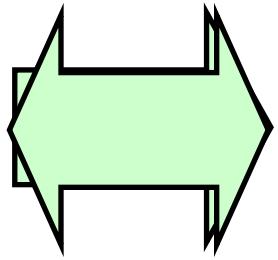
```
public class TestClassJPA {

    public static void main(String[] args) throws Exception {
        EntityManagerFactory entityManagerFactory = null;
        EntityManager entityManager = null;
        try {
            entityManagerFactory = Persistence.createEntityManagerFactory("TestJpa");
            entityManager = entityManagerFactory.createEntityManager();

            Personne pers = entityManager.find(Personne.class, 1 );
            System.out.println(pers );

       } finally {


      System.out.println( "- Insertion d'une nouvelle Personne" );
      Personne newPers = new Personne( "Ali", "Hicham" );
      entityManager.persist(newPers);



   System.out.println( "- Lecture de tous les personnes" );
   List<Personne> personnes = entityManager.createQuery( "from Personne",Personne.class)
                           .getResultList();
   for (Personne perss : personnes) {
       System.out.println(perss);
   }
```

Relations

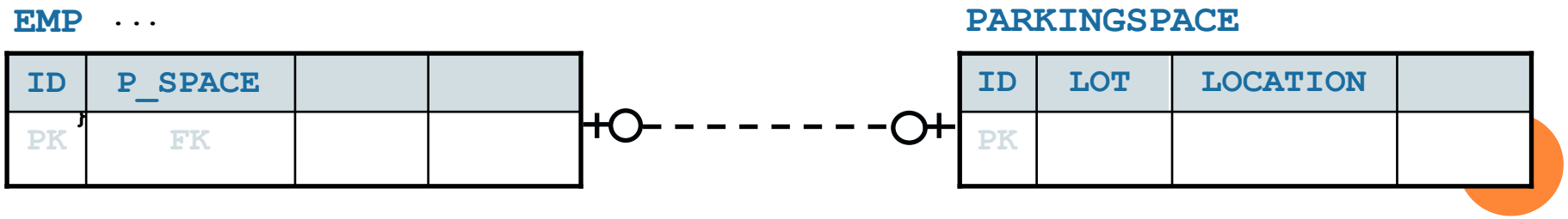
- 4 types de relations à définir entre les entités de la JPA:
  - One to One
  - Many to One (One to Many)
  - Many to Many

Relationships: One to One

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

  @Id

  private int id;

  @OneToOne
 @JoinColumn(name="P_SPACE")

  private ParkingSpace space;


  // getters & setters
  ...
}
```



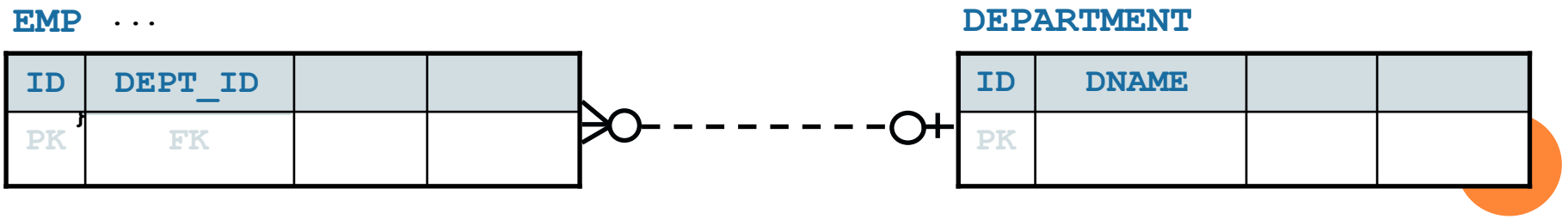
```
@Entity
public class ParkingSpace {

  @Id
  private int id;

  private int lot;

  private String location;

  @OneToOne(mappedBy="space")
  private Employee emp;

  // getters & setters
  ...

}
```



Relationship: One to Many

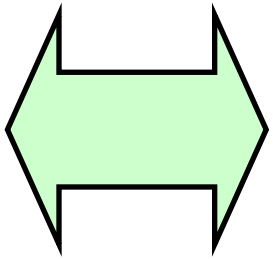
```java
@Entity
@Table(name="EMP")
public class Employee {

  @Id

  private int id;
  @ManyToOne
  @JoinColumn(name="DEPT_ID")


  private Department d;


  // getters & setters
  ...
}
```

```java
@Entity
public class Department {

  @Id
  private int id;

  private String dname;

  @OneToMany(mappedBy="d")
  private Collection<Employee> emps;

  // getters & setters
  ...
}
```



Relationships: Many to Many

```
@Entity
@Table(name="EMP")
public class Employee {
@Id  private int id;
    private String name;
@ManyToMany
@JoinTable(name="EMP_PROJ",
    joinColumns=
    @JoinColumn(name="EMP_ID"),
    inverseJoinColumns=
    @JoinColumn(name="PROJ_ID"))

private Collection<Project> p;
}
```



```
@Entity
public class Project {

  @Id
  private int id;

  private String name;

  @ManyToMany(mappedBy="p")
  private Collection<Employee> e;

  // getters & setters
  ...

}
```