

# JPA

---

- Introduction
- Les entités
  - Le mapping entre une entité et une table
  - Le mapping de propriété complexe
  - Mapper une entité sur plusieurs tables
  - Utilisation d'objets embarqués dans les entités
  - Fichier de configuration du mapping
  - Utilisation du bean entité
- EntityManager
  - Le fichier persistence.xml
  - La gestion des transactions hors Java EE
  - La gestion des relations entre tables dans le mapping
  - Les callbacks d'événements

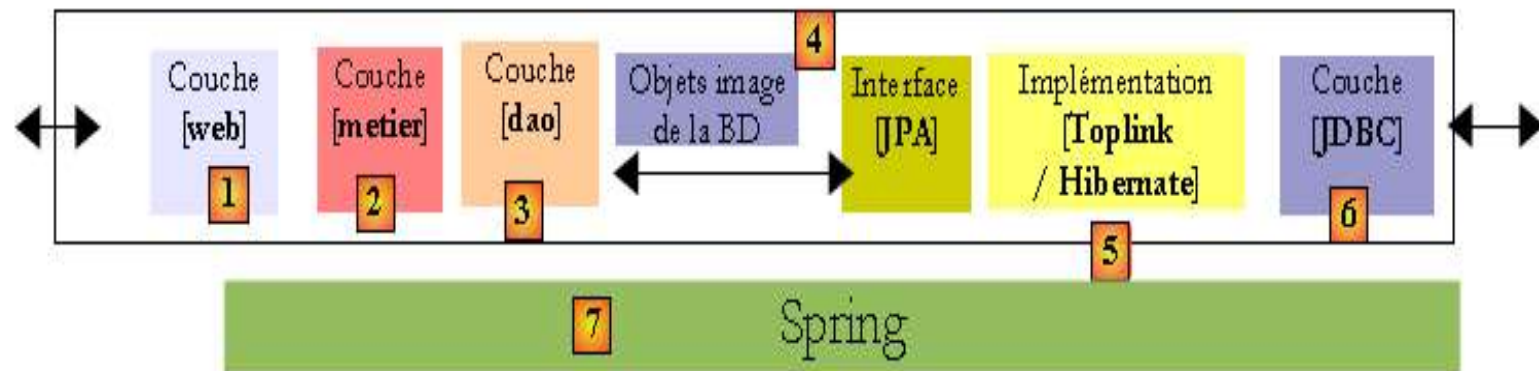
# JPA : Introduction

---

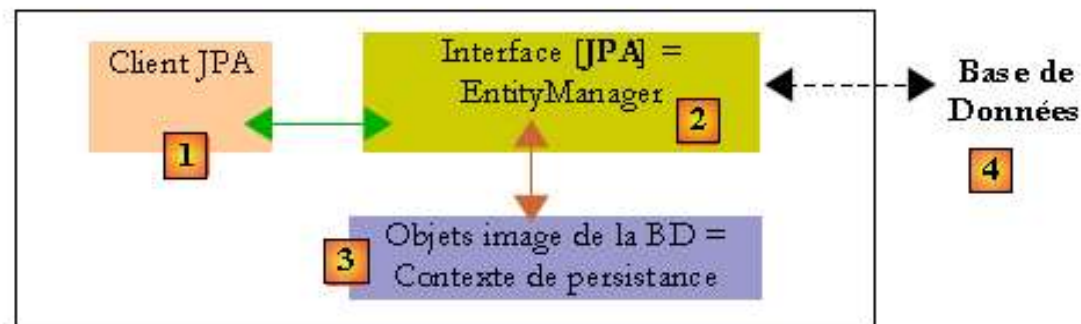
- JPA : Java Persistence API : outil ORM ;
- JPA c'est une API développée dans le cadre d'EJB 3.0 ;
- JPA ne se limite pas aux EJB 3 ;
- JPA ne requiert aucune ligne de code mettant en œuvre l'API JDBC ;
- JPA propose un langage d'interrogation similaire à SQL mais utilisant des objets plutôt que des entités relationnelles de la base de données ;
- JPA se base sur les entités (des objets POJO) et sur le gestionnaire de ces entités (EntityManager) ;
- EntityManager est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

# JPA : Introduction

## ➤ Position du JPA dans une architecture 3-tiers :



## ➤ Le contexte de persistance d'une application :



# JPA : Introduction

- L'implémentation de référence de JPA est incluse dans le projet *Glassfish*, disponible à l'URL :

<https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html>

- Cette implémentation de référence repose sur l'outil TopLink d'Oracle dans sa version essential ;
- Pour obtenir les fichiers .jar de JPA il suffit d'exécuter la commande suivante : *java -jar* avec en paramètre le fichier jar téléchargé.

- Exemple : la commande *java -jar glassfish-persistence-installer-v2-b58g.jar* donne comme résultat :



Nom	Date de modificati...	Type
Capacité non spécifiée (5)		
3RD-PARTY-LICENSE.txt	27/08/2009 23:43	Document texte
LICENSE.txt	27/08/2009 23:43	Document texte
README	27/08/2009 23:43	Fichier
toplink-essentials.jar	27/08/2009 23:43	Archive WinRAR
toplink-essentials-agent.jar	27/08/2009 23:43	Archive WinRAR

## JPA : Les entités

---

- Les entités permettent d'encapsuler les données d'une ou plusieurs tables ;
- Les entités sont des simples *POJO* (*Plain Old Java Object*) ;
- Les entités utilisent les annotations pour mapper un objet JAVA vers une table de la base de données ;
- Un bean entité doit obligatoirement avoir un constructeur sans argument et la classe du bean doit obligatoirement être marquée avec l'annotation `@javax.persistence.Entity` ;
- L'annotation `@javax.persistence.Entity` possède un attribut optionnel nommé *name* qui permet de préciser le nom de l'entité dans les requêtes. Par défaut, ce nom est celui de la classe de l'entité.

## JPA : Les entités

---

- Le bean Entité doit respecter les deux règles suivantes :
  - il doit être annoté par `@javax.persistence.Entity` ;
  - il doit posséder au moins une propriété déclarer comme clé primaire avec l'annotation `@Id`.
- Le bean entity est composé de propriétés qui seront mappés sur les champs de la table de la base de données ;
- Chaque propriété encapsule les données d'un champ d'une table. Ces propriétés sont utilisables au travers des getter et des setter ;
- Une propriété particulière est la clé primaire qui est l'identifiant unique dans la base de données et aussi dans le *POJO*.

# JPA

---

## ➤ Introduction

### ➤ Les entités

### ➤ Le mapping entre une entité et une table

- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
- Utilisation du bean entité
- EntityManager
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre table dans le mapping
- Les callbacks d'événements

## JPA: Le mapping entre une entité et une table

---

- La description du mapping entre le bean entité et la table peut être fait de deux façons :
  - Utiliser des annotations ;
  - Utiliser un fichier XML de mapping.
- L'API propose plusieurs annotations pour supporter un mapping O/R assez complet ;



## JPA: Le mapping entre une entité et une table

Annotation	Rôle
<code>@javax.persistence.Table</code>	Préciser le nom de la table concernée par le mapping
<code>@javax.persistence.Column</code>	Associé à un getter, il permet d'associer un champ de la table à la propriété
<code>@javax.persistence.Id</code>	Associé à un getter, il permet d'associer un champ de la table à la propriété en tant que clé primaire
<code>@javax.persistence.GeneratedValue</code>	Demander la génération automatique de la clé primaire au besoin
<code>@javax.persistence.Basic</code>	Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
<code>@javax.persistence.Transient</code>	Demander de ne pas tenir compte du champ lors du mapping

# JPA: Le mapping entre une entité et une table

## @Table

---

➤ L'entité @Table possède les attributs suivants :

Attribut	Rôle
name	Nom de la table
Catalog	Catalogue de la table
schema	Schéma de la table
uniqueConstraints	Contraintes d'unicité sur une ou plusieurs colonnes

# JPA: Le mapping entre une entité et une table

## @Column

➤ L'entité @Column possède les attributs suivants :

Attribut	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi-table
unique	Indique si la colonne est unique
nullable	Indique si la colonne est nullable
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type insert
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type update
columnDefinition	Précise le DDL de définition de la colonne
length	Indique la taille d'une colonne de type chaîne de caractères
precision	Indique la taille d'une colonne de type numérique
scale	Indique la précision d'une colonne de type numérique

## JPA: Le mapping entre une entité et une table

### @GeneratedValue

- L'annotation `@GeneratedValue` possède les attributs suivants :

Attribut	Rôle
strategy	Précise le type de générateur à utiliser :TABLE, SEQUENCE, IDENTITY ou AUTO. La valeur par défaut est AUTO
generator	Nom du générateur à utiliser

- Le type AUTO est le plus généralement utilisé : il laisse l'implémentation générer la valeur de la clé primaire ;
- Le type IDENTITY utilise un type de colonne spécial de la base de données.

# JPA: Le mapping entre une entité et une table

## @GeneratedValue

Exemple :

```
Personne.java X
package emsi.formation.jpa;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    private String prenom;
    private String nom;
    private static final long serialVersionUID = 1L;
    public Personne() {
        super();
    }
    public int getId() {
        return this.id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getPrenom() {
        return this.prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    public String getNom() {
        return this.nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
}
```

## JPA: Le mapping entre une entité et une table

### @GeneratedValue

- Le type *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` ;
- L'annotation `@TableGenerator` possède plusieurs attributs :

Attribut	Rôle
name	Nom identifiant le TableGenerator : il devra être utilisé comme valeur dans l'attribut generator de l'annotation <code>@GeneratedValue</code>
table	Nom de la table utilisé
catalog	Nom du catalogue utilisé
schema	Nom du schéma utiliser
pkColumnName	Nom de la colonne qui précise la clé primaire à générer
valueColumnName	Nom de la colonne qui contient la valeur de la clé primaire générée
pkColumnValue	
allocationSize	Valeur utilisée lors de l'incrémentement de la valeur de la clé primaire
uniqueConstraints	

## JPA: Le mapping entre une entité et une table

### @GeneratedValue

---

- Le type *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` ;
- L'annotation `@SequenceGenerator` possède plusieurs attributs :

Attribut	Rôle
name	Nom identifiant le <i>SequenceTableGenerator</i> : il devra être utilisé comme valeur dans l'attribut <code>generator</code> de l'annotation <code>@GeneratedValue</code>
sequenceName	Nom de la séquence dans la base de données
initialValue	Valeur initiale de la séquence
allocationSize	Valeur utilisée lors l'incrémentement de la valeur de la séquence

# JPA: Le mapping entre une entité et une table

## @GeneratedValue

- Exemple :

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@SequenceGenerator(name = "PERSONNE_SEQUENCE", sequenceName = "PERSONNE_SEQ")
public class Personne implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PERSONNE_SEQUENCE")
    private int id;
```



## JPA: Le mapping entre une entité et une table

### @GeneratedValue

---

➤ Le modèle de base de données relationnelle permet la définition d'une clé primaire composée de plusieurs colonnes. JPA propose deux façons de gérer ce cas de figure :

- L'annotation `@javax.persistence.IdClass`
- L'annotation `@javax.persistence.EmbeddedId`

➤ L'annotation `@IdClass` s'utilise avec une classe qui va encapsuler les propriétés qui composent la clé primaire. Cette classe doit obligatoirement :

- Être *sérialisable* ;
- Posséder un constructeur sans argument ;
- Fournir une implémentation dédiée des méthodes `equals()` et `hashCode()`.

# JPA: Le mapping entre une entité et une table

## @GeneratedValue

Exemple : La clé primaire est composée des champs nom et prenom (exemple théorique qui présume que deux personnes ne peuvent avoir le même nom et prénom)

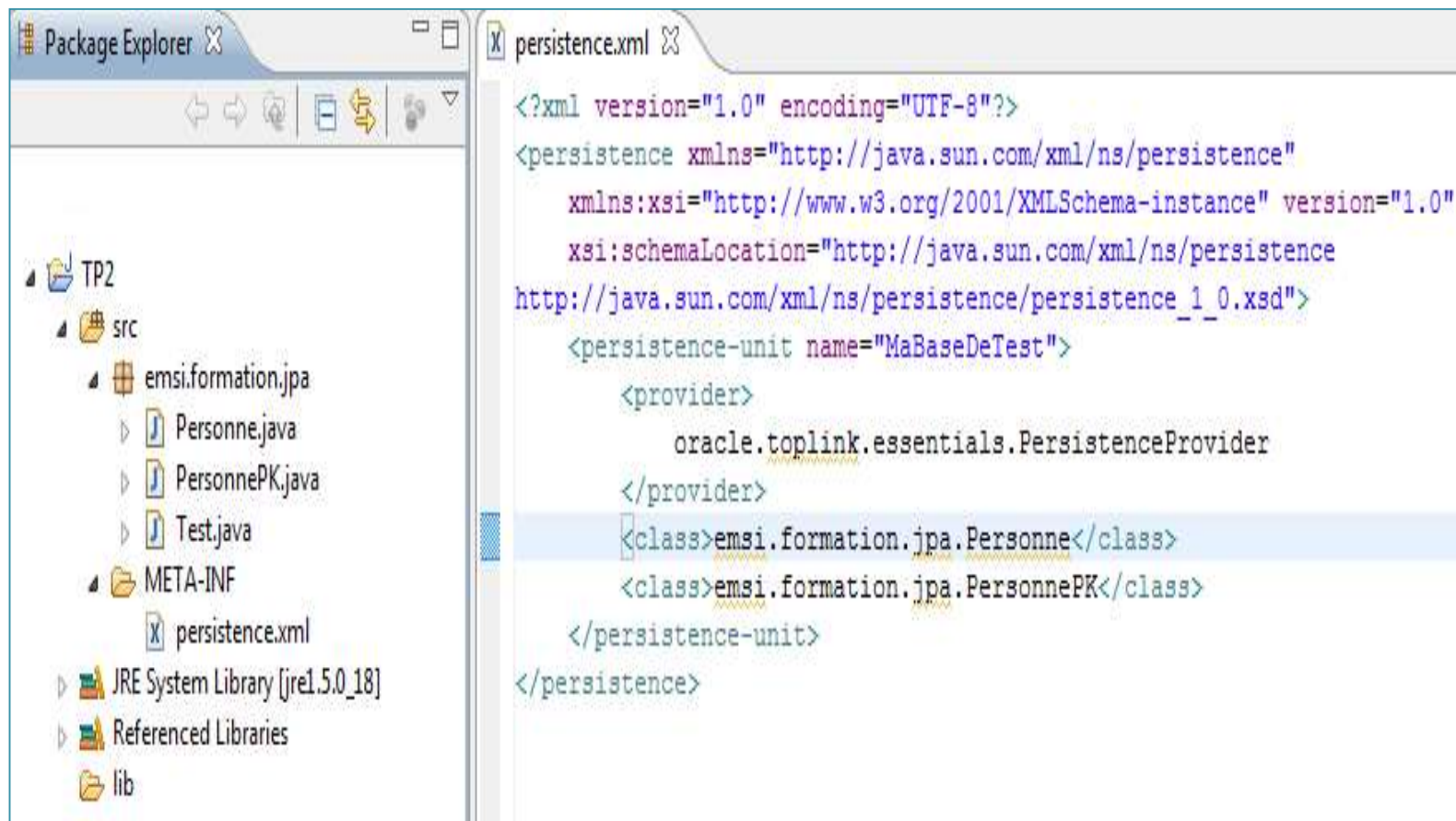
```
PersonnePK.java X
package emsi.formation.jpa;
public class PersonnePK implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private String nom;
    private String prenom;
    public PersonnePK() {
    }
    public PersonnePK(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    public String getNom() {
        return this.nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
}
```

```
PersonnePK.java X
public boolean equals(Object obj) {
    boolean resultat = false;
    if (obj == this) {
        resultat = true;
    } else {
        if (!(obj instanceof PersonnePK)) {
            resultat = false;
        } else {
            PersonnePK autre = (PersonnePK) obj;
            if (!nom.equals(autre.nom)) {
                resultat = false;
            } else {
                if (prenom != autre.prenom) {
                    resultat = false;
                } else {
                    resultat = true;
                }
            }
        }
    }
    return resultat;
}
public int hashCode() {
    return (nom + prenom).hashCode();
}
}
```

# JPA: Le mapping entre une entité et une table

## @GeneratedValue

- Il est nécessaire de définir la classe de la clé primaire dans le fichier de configuration *persistence.xml* :



## JPA: Le mapping entre une entité et une table

### @GeneratedValue

- L'annotation `@IdClass` possède un seul attribut :

Attribut	Rôle
Class	Classe qui encapsule la clé primaire composée

- Il faut utiliser l'annotation `@IdClass` sur la classe de l'entité ;
- Il est nécessaire de marquer chacune des propriétés de l'entité qui compose la clé primaire avec l'annotation `@Id` ;
- Ces propriétés doivent avoir le même nom dans l'entité et dans la classe qui encapsule la clé primaire ;

# JPA: Le mapping entre une entité et une table

## @GeneratedValue

### Exemple :

```
package emsi.formation.jpa;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(PersonnePK.class)
public class Personne implements Serializable {
    private String prenom;
    private String nom;
    private int taille;
    private static final long serialVersionUID = 1L;

    public Personne() {
        super();
    }

    @Id
    public String getPrenom() {
        return this.prenom;
    }
}
```

```
public void setPrenom(String prenom) {
    this.prenom = prenom;
}

@Id
public String getNom() {
    return this.nom;
}

public void setNom(String nom) {
    this.nom = nom;
}

public int getTaille() {
    return this.taille;
}

public void setTaille(int taille) {
    this.taille = taille;
}
}
```

## JPA: Le mapping entre une entité et une table `@GeneratedValue`

---

- Il n'est pas possible de demander la génération automatique d'une clé primaire composée ;
- La classe de la clé primaire est utilisée notamment lors de la recherche ;

Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");  
Personne personne = entityManager.find(Personne.class, clePersonne);
```



# JPA: Le mapping entre une entité et une table

## @GeneratedValue


- L'annotation `@EmbeddedId` s'utilise avec l'annotation `@javax.persistence.Embeddable`

```
PersonnePK.java X
package emsi.formation.jpa;
import javax.persistence.Embeddable;
@Embeddable
public class PersonnePK implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private String nom;
    private String prenom;
    public PersonnePK() {
    }
    public PersonnePK(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    public String getNom() {
        return this.nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
}
```

```
PersonnePK.java X
    public boolean equals(Object obj) {
        boolean resultat = false;
        if (obj == this) {
            resultat = true;
        } else {
            if (!(obj instanceof PersonnePK)) {
                resultat = false;
            } else {
                PersonnePK autre = (PersonnePK) obj;
                if (!nom.equals(autre.nom)) {
                    resultat = false;
                } else {
                    if (prenom != autre.prenom) {
                        resultat = false;
                    } else {
                        resultat = true;
                    }
                }
            }
        }
        return resultat;
    }
    public int hashCode() {
        return (nom + prenom).hashCode();
    }
}
```

# JPA: Le mapping entre une entité et une table

## @GeneratedValue



```
PersonnePK.java  Personne.java X
package emsi.formation.jpa;
import java.io.Serializable;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
@Entity
public class Personne implements Serializable {
    @EmbeddedId
    private PersonnePK clePrimaire;
    private int taille;
    private static final long serialVersionUID = 1L;
    public Personne() {
        super();
    }
    public PersonnePK getClePrimaire() {
        return clePrimaire;
    }
    public void setClePrimaire(PersonnePK clePrimaire) {
        this.clePrimaire = clePrimaire;
    }
    public int getTaille() {
        return taille;
    }
    public void setTaille(int taille) {
        this.taille = taille;
    }
}
```



## JPA: Le mapping entre une entité et une table

### @GeneratedValue


---

- L'annotation `@AttributeOverrides` est une collection d'attributs `@AttributeOverride` ;
- Ces annotations permettent de ne pas avoir à utiliser l'annotation `@Column` dans la classe de la clé ou de modifier les attributs de cette annotation dans l'entité qui la met en œuvre ;
- L'annotation `@AttributeOverride` possède plusieurs attributs :

Attribut	Rôle
name	Précise le nom de la propriété de la classe imbriquée
column	Précise la colonne de la table à associer à la propriété

# JPA: Le mapping entre une entité et une table

## @GeneratedValue



```
PersonnePK.java  Personne.java X
package emsi.formation.jpa;
import java.io.Serializable;
import javax.persistence.AttributeOverrides;
import javax.persistence.AttributeOverride;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Column;
@Entity
public class Personne implements Serializable {
    @EmbeddedId
    @AttributeOverrides( {
        @AttributeOverride(name = "nom", column = @Column(name = "NOM")),
        @AttributeOverride(name = "prenom", column = @Column(name = "PRENOM")) })
    private PersonnePK clePrimaire;
```

## JPA: Le mapping entre une entité et une table

### *@Transient*

---

- Par défaut, toutes les propriétés sont mappées sur la colonne correspondante dans la table.
- L'annotation `@javax.persistence.Transient` permet d'indiquer au gestionnaire de persistance d'ignorer cette propriété.

# JPA: Le mapping entre une entité et une table

## *@Basic*

---

- L'annotation `@javax.persistence.Basic` représente la forme de mapping la plus simple.
- `@Basic` est optionnelle et est celle utilisée par défaut ;
- Ce mapping concerne :
  - les types primitifs ;
  - les wrappers de type primitifs ;
  - les tableaux de ces types ;
  - les types `java.math.BigInteger` ;
  - `java.math.BigDecimal` ;
  - `java.util.Date` ;
  - `java.util.Calendar` ;
  - `java.sql.Date` ;
  - `java.sql.Time` et `java.sql.Timestamp`.

# JPA: Le mapping entre une entité et une table

## @Basic

➤ L'annotation `@Basic` possède plusieurs attributs :

Attribut	Rôle
fetch	<p>Permet de préciser comment la propriété est chargée selon deux mode :</p> <ul style="list-style-type: none"><li>▪ LAZY : la valeur est chargée uniquement lors de son utilisation ;</li><li>▪ EAGER : la valeur est toujours chargée (valeur par défaut).</li></ul> <p>Cette fonctionnalité permet de limiter la quantité de données obtenues par une requête.</p>
optionnal	Indique que la colonne est nullable.

# JPA: Le mapping entre une entité et une table

## @Basic

➤ Exemple :

```
Personne.java X
package emsi.formation.jpa;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;
    private static final long serialVersionUID = 1L;

    public int getId() {
        return id;
    }
}
```

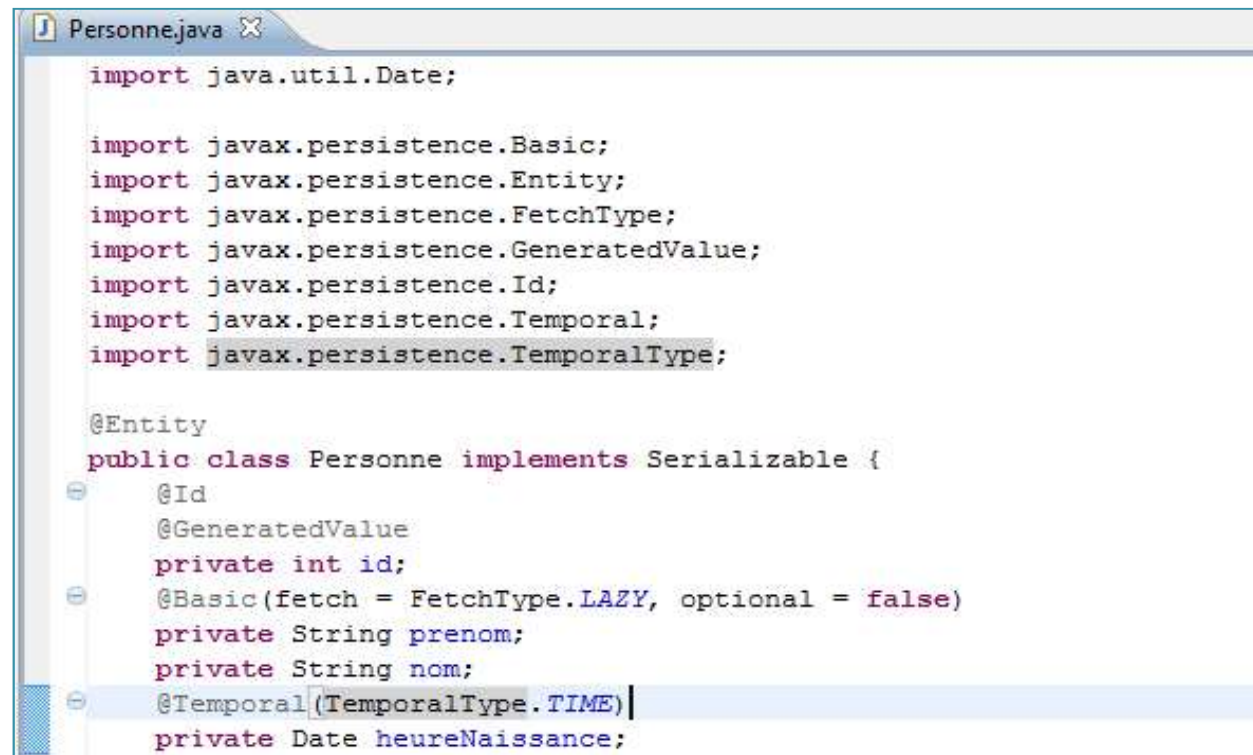
# JPA: Le mapping entre une entité et une table

## @Temporal

➤ L'annotation `@javax.persistence.Temporal` permet de fournir des informations complémentaires sur la façon dont les propriétés encapsulant des données temporelles (Date et Calendar) ;

➤ La valeur par défaut est *timestamp* ;

➤ Exemple :



```
Personne.java X
import java.util.Date;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;
    @Temporal(TemporalType.TIME)
    private Date heureNaissance;
```

# JPA

---

## ➤ Introduction

- Les entités
- Le mapping entre une entité et une table
- **Le mapping de propriété complexe**
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
- Utilisation du bean entité
- EntityManager
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre table dans le mapping
- Les callbacks d'événements



# JPA: Le mapping de propriété complexe

---

- JPA permet de mapper les champs de types Blob et Clob ;
- L'annotation `@javax.persistence.Lob` permet mapper une propriété sur une colonne de type Blob ou Clob selon le type de la propriété :
  - Blob pour les tableaux de byte ou Byte ou les objets sérializables ;
  - Clob pour les chaînes de caractères et les tableaux de caractères char ou Char.

# JPA: Le mapping de propriété complexe @Lob

➤ exemple :

```
Personne.java X
package emsi.formation.jpaa;
import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;
import com.sun.imageio.plugins.jpeg.JPEG;
@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;
    @Lob
    @Basic(fetch = FetchType.LAZY)
    private JPEG photo;
    private static final long serialVersionUID = 1L;
}
```

```
Personne.java X
public Personne() {
    super();
}
public int getId() {return this.id;}
public void setId(int id) {
    this.id = id;
}
public JPEG getPhoto() {return photo;}
public void setPhoto(JPEG photo) {this.photo = photo;}
public String getPrenom() {return prenom;}
public void setPrenom(String prenom) {
    this.prenom = prenom;
}
public String getNom() {return nom;}
public void setNom(String nom) {this.nom = nom;}
}
```

# JPA: Le mapping de propriété complexe

## @Enumerated

- L'annotation `@javax.persistence.Enumerated` permet d'associer une propriété de type énumération à une colonne de la table sous la forme d'un numérique ou d'une chaîne de caractères ;
- Cette forme est précisée en paramètre de l'annotation grâce à l'énumération `EnumType` qui peut avoir comme valeur `EnumType.ORDINAL` (valeur par défaut) ou `EnumType.STRING`.

➤ exemple :

```
public enum Genre {  
    HOMME,  
    FEMME,  
    INCONNU  
}
```

```
@Entity  
public class Personne implements Serializable {  
    @Id  
    @GeneratedValue  
    private int id;  
  
    @Basic(fetch = FetchType.LAZY, optional = false)  
    private String prenom;  
  
    private String nom;  
  
    @Enumerated(EnumType.STRING)  
    private Genre genre;
```

# JPA

---

## ➤ Introduction

- Les entités
- Le mapping entre une entité et une table
- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
- Utilisation du bean entité
- EntityManager
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre table dans le mapping
- Les callbacks d'événements

# JPA: Mapper une entité sur plusieurs tables

- L'annotation `@javax.persistence.SecondaryTable` permet de préciser qu'une autre table sera utilisée dans le mapping ;
- Pour utiliser cette fonctionnalité, la seconde table doit posséder une jointure entre sa clé primaire et une ou plusieurs colonnes de la première table.
- L'annotation `@SecondaryTable` possède plusieurs attributs :

Attribut	Rôle
Name	Nom de la table
Catalogue	Nom du catalogue
Schema	Nom du schéma
pkJoinColumns	Collection des clés primaire de la jointure sous la forme d'annotations de type <code>@PrimaryKeyJoinColumn</code>
uniqueConstraints	

# JPA: Mapper une entité sur plusieurs tables

## @SecondaryTable

- L'annotation `@PrimaryKeyJoinColumn` permet de préciser une colonne qui compose la clé primaire de la seconde table et entre dans la jointure avec la première table. Elle possède plusieurs attributs :

Attribut	Rôle
<code>name</code>	Nom de la colonne
<code>referencedColumnName</code>	Nom de la colonne dans la première table (obligatoire si les noms de colonnes sont différents entre les deux tables)
<code>columnDefinition</code>	

- Il est nécessaire pour chaque propriété de l'entité qui est mappée sur la seconde table de renseigner le nom de la table dans l'attribut `table` de l'annotation `@Column` ;



# JPA: Mapper une entité sur plusieurs tables

## @SecondaryTable

Exemple :

Database *mabasedetest* - Table *ADRESSE* running on *localhost*

Structure		Browse		SQL		Select		Insert			
Field		Type	Attributes	Null	Default	Extra	Action				
<input type="checkbox"/>	ID_ADRESSE	int(11)		No	0						
<input type="checkbox"/>	RUE	varchar(50)		No							
<input type="checkbox"/>	CODEPOSTAL	varchar(7)		No							
<input type="checkbox"/>	VILLE	varchar(50)		No							

```
PersonneAdresse.java
package emsi.for

import java.io.Serializable;
import javax.persistence.*;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@SecondaryTable(name = "ADRESSE", pkJoinColumns = { @PrimaryKeyJoinColumn(name = "ID_ADRESSE") })
public class PersonneAdresse implements Serializable {

    @Id
    @GeneratedValue
    private int id;

    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;

    @Column(name = "RUE", table = "ADRESSE")
    private String rue;

    @Column(name = "CODEPOSTAL", table = "ADRESSE")
    private String codePostal;

    @Column(name = "VILLE", table = "ADRESSE")
    private String ville;

    private static final long serialVersionUID = 1L;
}
```

# JPA: Mapper une entité sur plusieurs tables

## @SecondaryTables

➤ Si le mapping d'une entité utilise plusieurs tables, il faut utiliser l'annotation `@javax.persistence.SecondaryTables` qui est une collection d'annotation `@SecondaryTable` ;

➤ Exemple :

```
PersonneAdresse.java X
package emsi.formation.jpa;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
import javax.persistence.SecondaryTables;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@SecondaryTables( {
    @SecondaryTable(name = "ADRESSE", pkJoinColumns = { @PrimaryKeyJoinColumn(name = "ID_ADRESSE") }),
    @SecondaryTable(name = "INFO_PERS", pkJoinColumns = { @PrimaryKeyJoinColumn(name = "ID_INFO_PERS") }) })
public class PersonneAdresse implements Serializable {
    @Id
    @GeneratedValue
    private int id;
```

➤ **Travaux pratiques** : Réaliser le **TP4** relatif à l'API JPA.



# JPA

---

## ➤ Introduction

- Les entités
- Le mapping entre une entité et une table
- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- **Utilisation d'objets embarqués dans les entités**
- Fichier de configuration du mapping
- Utilisation du bean entité
- EntityManager
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre table dans le mapping
- Les callbacks d'événements

# JPA: Utilisation d'objets embarqués dans les entités

---

- JPA permet d'utiliser dans les entités des objets Java qui ne sont pas des entités mais qui sont agrégés dans l'entité et dont les propriétés seront mappées sur les colonnes correspondantes dans la table ;
- La mise en œuvre de cette fonctionnalité est similaire à celle utilisée avec l'annotation `@EmbeddedId` pour les clés primaires composées ;
- La classe embarquée est un simple *POJO* qui doit être marquée avec l'annotation `@javax.persistence.Embeddable` ;
- *Le champs embarqué doit être annoté par `@Embedded`.*
- **Travaux pratiques** : Réaliser le **TP5** relatif à l'API JPA.

# JPA

---

## ➤ Introduction

- Les entités
- Le mapping entre une entité et une table
- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités

## ➤ Fichier de configuration du mapping

- Utilisation du bean entité
- EntityManager
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre tables dans le mapping
- Les callbacks d'événements

# JPA: Fichier de configuration du mapping

---

- JPA offre la possibilité de définir le mapping dans un fichier XML ;
- Le nom du fichier est par défaut *orm.xml* stocké dans le répertoire META-INF ;
- L'élément racine est le tag `<entity-mappings>` ;
- Pour chaque entité, il faut utiliser un tag fils `<entity>`. Ce tag possède deux attributs :
  - *Class* : qui permet préciser le nom pleinement qualifié de la classe de l'entité ;
  - *Access* : qui permet de préciser le type d'accès aux données (*PROPERTY* pour un accès via les getter/setter ou *FIELD* pour un accès via les champs).
- La déclaration de la clé primaire se fait dans un tag `<id>` fils d'un tag `<attributes>`.
- Le tag `<id>` possède un attribut nommé *name* qui permet de préciser le nom du champ qui est la clé primaire.

# JPA: Utilisation du bean entité

---

- Il est conseillé de maintenir le rôle du bean entity au transfert de données : il faut éviter de lui ajouter des méthodes métier ;
- Les bean entity peuvent être utilisées dans l'échange entre le client et le serveur ;
- Toutes les actions de persistance sur ces objets sont réalisées grâce à un objet dédié de l'API : l'EntityManager.
- Un contexte de persistance (persistence context) est un ensemble d'entités géré par un EntityManager ;
- Les entités peuvent ainsi être de deux types Gérée et Non gérée.
- Lorsqu'un contexte de persistance est fermé, toutes les entités du contexte deviennent non gérées.

# JPA: Utilisation du bean entité

---

➤ Il existe deux types de contexte de persistance :

- *Transaction-scoped* : le contexte est ouvert pendant toute la durée d'une transaction. La fermeture de la transaction entraîne la fermeture du contexte.
  - Ce type de contexte n'est utilisable que dans le cadre de l'utilisation dans un conteneur qui va assurer la prise en charge de la transaction.
- *Extended persistence* : le contexte reste ouvert après la fermeture de la transaction

# JPA

---

## ➤ Introduction

- Les entités
- Le mapping entre une entité et une table
- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
- Utilisation du bean entité
- **EntityManager**
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre tables dans le mapping
- Les callbacks d'événements



# JPA: EntityManager

---

- Les interactions entre la base de données et les beans entité sont assurées par un objet de type *javax.persistence.EntityManager* ;
- *EntityManager* permet de lire, rechercher des données et de les mettre à jour (ajout, modification, suppression) ;
- *EntityManager* assure les interactions avec le gestionnaire de transactions ;
- *EntityManager* gère un ensemble défini de beans entité nommé *persistence unit* ;
- La définition d'un *persistence unit* est assurée dans un fichier de description nommé *persistence.xml*.

# JPA: EntityManager

---

➤ Dans un environnement Java SE, comme par exemple dans Tomcat ou dans une application de type client lourd, l'instanciation d'un objet de type *EntityManager* doit être codée ;

- il faut utiliser une fabrique de type *EntityManagerFactory* ;
- Cette fabrique propose la méthode *createEntityManager()* pour obtenir une instance ;
- Il faut utiliser la méthode *close()* de la fabrique une fois que cette dernière n'a plus d'utilité pour libérer les ressources.

➤ Dans un environnement Java EE, on peut utiliser l'IOC pour obtenir un objet de l'*EntityManager* ou obtenir une fabrique de type *EntityManagerFactory* qui sera capable de créer l'objet ;

- L'annotation *@javax.persistence.PersistenceUnit* sur un champ de type *EntityManagerFactory* permet d'injecter une fabrique ;
- Cette annotation possède un attribut *unitName* qui précise le nom de l'unité de persistance.

# JPA: EntityManager

## ➤ Exemple :

Exemple :

```
@PersistenceUnit(unitName="MaBaseDeTestPU")  
private EntityManagerFactory factory;
```

- Il est possible d'utiliser la fabrique pour obtenir un objet de type *EntityManager*.
- Pour associer ce contexte à la transaction courante, il faut utiliser la méthode *joinTransaction()* ;
- La méthode *close()* est automatiquement appelée par le conteneur : il ne faut pas utiliser cette méthode dans un conteneur sinon une exception de type *IllegalStateException* est levée.

# JPA: EntityManager

- L'annotation `@javax.persistence.PersistenceContext` sur un champ de type `EntityManager` permet d'injecter un contexte de persistance ;
- Cette annotation possède un attribut `unitName` qui précise le nom de l'unité de persistance ;

## ➤ Exemple :

```
GestionDeStockBean.java X
import javax.persistence.*;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class GestionDeStockBean implements GestionDeStock {

    @PersistenceContext
    EntityManager em;

    public void ajouter(Produit produit) {
        em.persist(produit);
    }

    public Produit rechercherProduit(String id) {
        return em.find(Produit.class, id);
    }

    public List<Produit> listerTousLesProduits() {
        return em.createQuery(
            "SELECT p FROM Produit p ORDER BY p.quantiteEnStock")
            .getResultList();
    }
}
```

# JPA: EntityManager

---

➤ La méthode *contains()* :

Permet de savoir si une instance fournie en paramètre est gérée par le contexte. Dans ce cas, elle renvoie *true*, sinon elle renvoie *false* ;

➤ La méthode *close()* : le contexte de persistance est fermé. Force la synchronisation du contexte de persistance avec la base de données :

- si un objet du contexte n'est pas présent dans la base, il y est mis par une opération *SQL INSERT* ;
- si un objet du contexte est présent dans la base et qu'il a été modifié depuis qu'il a été lu, une opération *SQL UPDATE* est faite pour persister la modification ;
- si un objet du contexte a été marqué comme " supprimé " à l'issue d'une opération *remove* sur lui, une opération *SQL DELETE* est faite pour le supprimer de la base.

# JPA: EntityManager

---

➤ La méthode *clear()* :

permet de détacher toutes les entités gérées par le contexte. Dans ce cas, toutes les modifications apportées aux entités sont perdues ;

- il est préférable d'appeler la méthode *flush()* avant la méthode *clear()*.

➤ La méthode *flush()* :

Le contexte de persistance est synchronisé avec la base de données de la façon décrite pour *close()*.

# JPA: EntityManager

---

- Le mode de synchronisation est géré par les méthodes suivantes de l'interface EntityManager :

void setFlushMode(FlushModeType flushMode) : Il y a deux valeurs possibles pour flushmode :

- FlushModeType.AUTO (défaut): la synchronisation a lieu avant chaque requête SELECT faite sur la base ;
- FlushModeType.COMMIT : la synchronisation n'a lieu qu'à la fin des transactions sur la base.

FlushModeType getFlushMode(.) : rend le mode actuel de synchronisation



# JPA: EntityManager

## Insertion dans la BDD

➤ Pour insérer des données dans la BDD il faut :

- Instancier une occurrence de la classe de l'entité ;
- Initialiser les propriétés de l'entité ;
- Définir les relations de l'entité avec d'autres entités au besoin ;
- Utiliser la méthode *persist()* de l'EntityManager en passant en paramètre l'entité.

```
private void initEntityManager() {
    emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
    em = emf.createEntityManager();
}

private void closeEntityManager() {
    em.close();
    emf.close();
}

private void addPersons() {
    em.getTransaction().begin();
    Personne p1 = new Personne("prenom1", "nom1");
    Personne p2 = new Personne("prenom2", "nom2");
    em.persist(p1);
    em.persist(p2);
    em.getTransaction().commit();
}
```

# JPA: EntityManager

## Recherche des occurrences

---

➤ Pour effectuer des recherches de données, l'EntityManager propose deux mécanismes :

- La recherche à partir de la clé primaire ;
  - `find()` et `getReference()` ;
- La recherche à partir d'une requête utilisant une syntaxe dédiée.

➤ Exemple :

```
private Personne search(int id) {  
    Personne personne = em.find(Personne.class, id);  
    return personne;  
}
```

# JPA: EntityManager

## Recherche par requête

La recherche par requête repose sur des méthodes dédiées de la classe EntityManager :

- `createQuery()`, `createNamedQuery()` et `createNativeQuery()` ;
- et sur un langage de requête spécifique nommé JPQL (Java Persistence Query Language). Une requête JPQL est analogue à une requête SQL (utilisation des objets plutôt que des tables).

➤ Exemple :

```
private List<Personne> searchByName(String nom) {  
    Query query = em.createQuery("select p from Personne p where p.nom='"+ nom +"'");  
    return query.getResultList();  
}
```

# JPA: EntityManager

## Recherche par requête

---

➤ L'objet *Query* gère aussi des paramètres nommés dans la requête. Le nom de chaque paramètre est préfixé par « : » dans la requête. La méthode *setParameter()* permet de fournir une valeur à chaque paramètre ;

➤ Exemple :

```
private List<Personne> searchByName2(String nom) {  
    Query query = em.createQuery("select p from Personne p where p.nom=:nom");  
    query.setParameter("nom", nom);  
    return query.getResultList();  
}
```

# JPA: EntityManager

## Modifier une occurrence

---

- Pour modifier une entité existante dans la base de données, il faut :
  - Obtenir une instance de l'entité à modifier (par recherche sur la clé primaire ou l'exécution d'une requête)
  - Modifier les propriétés de l'entité
  - Selon le mode de synchronisation des données de l'EntityManager, il peut être nécessaire d'appeler la méthode `flush()` explicitement

### ➤ Exemple :

```
private void updatePerson(int i) {  
    em.getTransaction().begin();  
    Personne personne=em.find(Personne.class, i);  
    personne.setNom("Nom modifié");  
    em.flush();  
    em.getTransaction().commit();  
}
```

# JPA: EntityManager merge et remove

## ➤ Exemple pour merge :

```
private void merge(int i) {  
    em.getTransaction().begin();  
    Personne personne=em.find(Personne.class, i);  
  
    Personne personne2=new Personne();  
    personne2.setId(personne.getId());  
    personne2.setNom("Nom Remodifié");  
    em.merge(personne2);  
    em.getTransaction().commit();  
}
```

## ➤ Exemple pour remove :

```
private void remove(int i) {  
    em.getTransaction().begin();  
    Personne personne=em.find(Personne.class, i);  
    em.remove(personne);  
    em.getTransaction().commit();  
}
```

# JPA

---

- Introduction
  - Les entités
  - Le mapping entre une entité et une table
  - Le mapping de propriété complexe
  - Mapper une entité sur plusieurs tables
  - Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
  - Utilisation du bean entité
  - EntityManager
- **Le fichier persistence.xml**
  - La gestion des transactions hors Java EE
  - La gestion des relations entre tables dans le mapping
  - Les callbacks d'événements



# JPA: Le fichier persistence.xml

---

- Ce fichier persistence.xml contient la configuration de base pour le mapping notamment en fournissant les informations sur la connexion à la base de données à utiliser ;
- Le fichier persistence.xml doit être stocké dans le répertoire *META-INF*;
- La racine du document XML du fichier *persistence.xml* est le tag `<persistence>` ;
  - Il contient un ou plusieurs tags `<persistence-unit>` ;
  - Ce tag possède deux attributs : *name* (obligatoire) qui précise le nom de l'unité et qui servira à y faire référence et *transaction-type* (optionnel) qui précise le type de transaction utilisée (ceci dépend de l'environnement d'exécution : Java SE ou Java EE).

# JPA: Le fichier persistence.xml

➤ Le tag `<persistence-unit>` peut avoir les tags fils suivants :

Tag	Rôle
<code>&lt;description&gt;</code>	Description purement informative de l'unité de persistance(optionnel)
<code>&lt;provider&gt;</code>	Nom pleinement qualifié d'une classe de type <code>javax.persistence.PersistenceProvider</code> (optionnel). Généralement fournie par le fournisseur de l'implémentation de l'API : une utilisation de ce tag n'est requise que pour des besoins spécifiques.
<code>&lt;jta-data-source&gt;</code>	Nom JNDI de la DataSource utilisée dans un environnement avec support de JTA (optionnel)
<code>&lt;non-jta-data-source&gt;</code>	Nom JNDI de la DataSource utilisée dans un environnement sans support de JTA (optionnel)
<code>&lt;mapping-file&gt;</code>	Précise un fichier de mapping supplémentaire (optionnel)
<code>&lt;jar-file&gt;</code>	Précise un fichier jar qui contient des entités à inclure dans l'unité de persistance : le chemin précisé est relatif par rapport au fichier persistence.xml (optionnel)

# JPA: Le fichier persistence.xml

Tag	Rôle
<class>	Précise une classe d'une entité qui sera incluse dans l'unité de persistence (optionnel)
<properties>	Fournir des paramètres spécifiques au fournisseur. Comme Java SE ne propose pas de serveur JNDI, c'est fréquemment via ce tag que les informations concernant la source de données sont définis (optionnel)
<exclude-unlisted-classes>	Inhibition de la recherche automatique des classes des entités (optionnel)

➤ L'ensemble des classes des entités qui compose l'unité de persistence peut être spécifié explicitement dans le fichier persistence.xml ou déterminé dynamiquement à l'exécution par recherche de toutes les classes possédant une annotation `@javax.persistence.Entity` ;

➤ Par défaut, la liste de classes explicites est complétée par la liste des classes issue de la recherche dynamique. Pour empêcher la recherche dynamique, il faut utiliser le tag `<exclude-unlisted-classes>`

# JPA: Le fichier persistence.xml

## ➤ Exemple :

A screenshot of a code editor window titled 'persistence.xml'. The code is XML and defines a JPA persistence unit. The XML structure is as follows: a root 'persistence' element containing a 'persistence-unit' element. The 'persistence-unit' has attributes 'name="TP2\_JPA"' and 'transaction-type="RESOURCE\_LOCAL"'. Inside the 'persistence-unit' is a 'provider' element with the value 'oracle.toplink.essentials.PersistenceProvider', a 'class' element with the value 'emsi.formation.jpa.Personne', and a 'properties' element. The 'properties' element contains five 'property' elements with the following details: 'toplink.jdbc.url' with value 'jdbc:mysql://localhost:3306/mabasedetest', 'toplink.jdbc.user' with value 'root', 'toplink.jdbc.driver' with value 'com.mysql.jdbc.Driver', 'toplink.jdbc.password' with value '', and 'toplink.ddl-generation' with value 'drop-and-create-tables'.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="TP2_JPA" transaction-type="RESOURCE_LOCAL">
    <provider>
      oracle.toplink.essentials.PersistenceProvider
    </provider>
    <class>emsi.formation.jpa.Personne</class>
    <properties>
      <property name="toplink.jdbc.url" value="jdbc:mysql://localhost:3306/mabasedetest" />
      <property name="toplink.jdbc.user" value="root" />
      <property name="toplink.jdbc.driver" value="com.mysql.jdbc.Driver" />
      <property name="toplink.jdbc.password" value="" />
      <property name="toplink.ddl-generation" value="drop-and-create-tables" />
    </properties>
  </persistence-unit>
</persistence>
```

# JPA

---

- Introduction
- Les entités
- Le mapping entre une entité et une table
- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
- Utilisation du bean entité
- EntityManager
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- La gestion des relations entre tables dans le mapping
- Les callbacks d'événements

# JPA: La gestion des transactions hors Java EE

- Le conteneur Java EE propose un support des transactions grâce à l'API JTA : c'est la façon standard de gérer les transactions par le conteneur ;
- Hors d'un tel conteneur, par exemple dans une application Java SE, les transactions ne sont pas supportées ;
- Dans le contexte Java SE, JPA propose une gestion des transactions grâce à l'interface *EntityManagerTransaction*. Cette interface propose plusieurs méthodes :

Méthode	Rôle
void begin()	Débuter la transaction
void commit()	Valider la transaction
void rollback()	Annuler la transaction
boolean isActive()	Déterminer si la transaction est active

# JPA: La gestion des transactions hors Java EE

- Pour obtenir une instance de la transaction, il faut utiliser la méthode *getTransaction()* de l'EntityManager ;
- La méthode *begin()* lève une exception de type *IllegalStateException* si une transaction est déjà active ;
- Les méthodes *commit()* et *rollback()* lèvent une exception de type *IllegalStateException* si aucune transaction n'est active ;

```
private void create() {  
    em.getTransaction().begin();  
    Personne p1 = new Personne("prenom1", "nom1", new Adresse("rue1", "codepostal1", "ville1") );  
    Personne p2 = new Personne("prenom2", "nom2", new Adresse("rue2", "codepostal2", "ville2"));  
    Personne[] ps = new Personne[] { p1, p2 };  
    for (Personne g : ps) {  
        em.persist(g);  
    }  
    em.getTransaction().commit();  
}
```



# JPA

---

- Introduction
- Les entités
- Le mapping entre une entité et une table
- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
- Utilisation du bean entité
- EntityManager
- Le fichier persistence.xml
- La gestion des transactions hors Java EE
- **La gestion des relations entre tables dans le mapping**
- Les callbacks d'événements

# JPA: La gestion des relations entre tables dans le mapping

---

➤ Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations :

- 1-1 (one-to-one) ;
- 1-n (one-to-many) ;
- n-1 (many-to-one) ;
- n-n (many-to-many).

➤ Travaux pratiques : Réaliser les TPs 6,7 et 8 relatif à l'API JPA.

# JPA: Les callbacks d'événements

---

➤ JPA permet de définir des callbacks qui seront appelés sur certains événements. Ces callbacks doivent être annotés avec une des annotations définies par JPA :

- `@javax.persistence.PrePersist` ;
- `@javax.persistence.PostPersist` ;
- `@javax.persistence.PostLoad` ;
- `@javax.persistence.PreUpdate` ;
- `@javax.persistence.PostUpdate` ;
- `@javax.persistence.PreRemove` ;
- `@javax.persistence.PostRemove`.



# Le Framework de développement: Spring

---

- Introduction
- Les versions de Spring
- Les modules de Spring

# Introduction

---

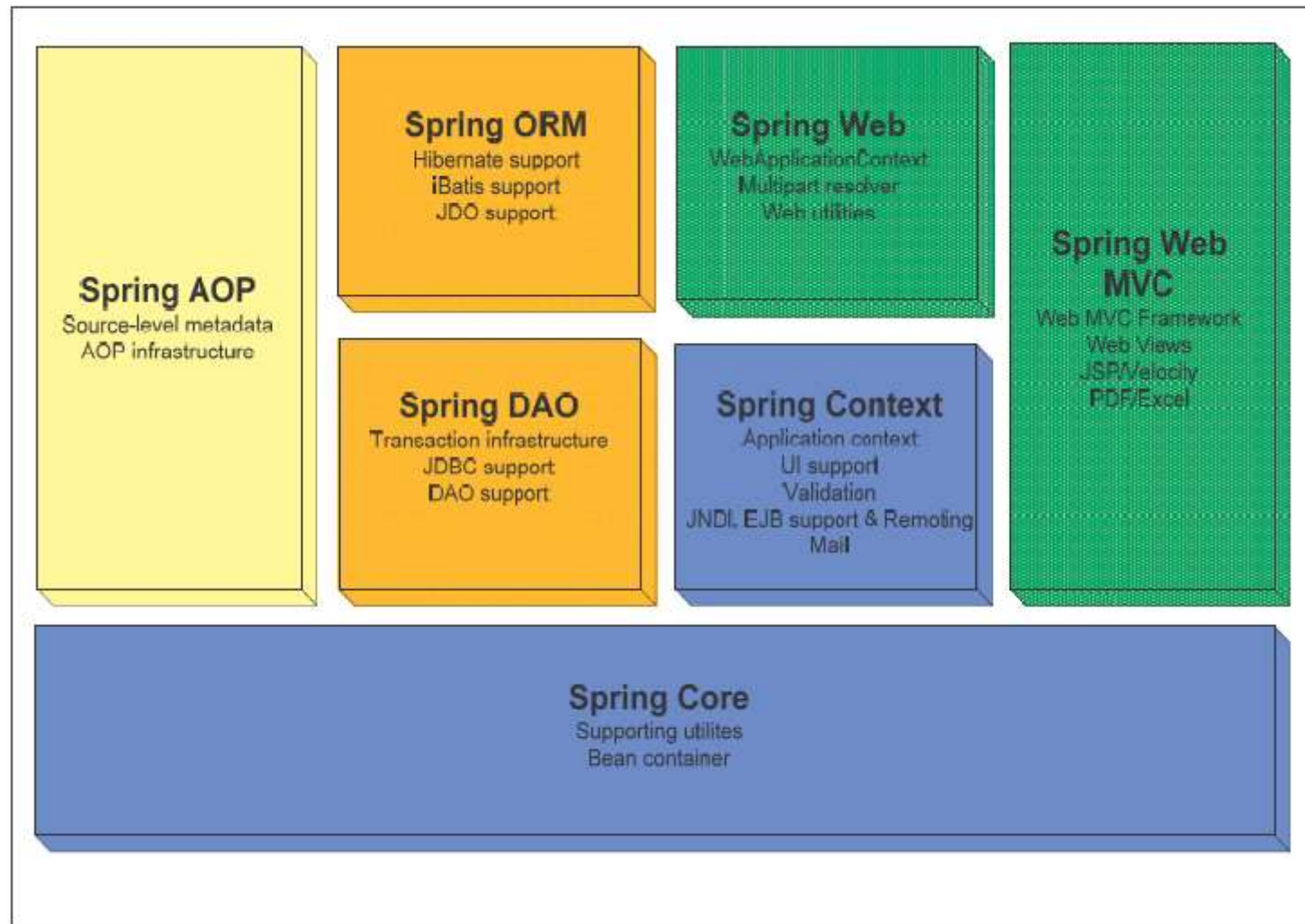
- Spring est un Framework développé en Java
- Spring est basé sur des concepts simples
- Spring a été créé par Rod Johnson en 2002
- Spring est un ensemble d'API modulaire
- Tests unitaires très faciles à réaliser.
- Notion de conteneur léger
- Nombreux modules optionnels permettant au développeur de se consacrer à ce que fait son application et non plus comment elle le fait

# Les versions de Spring

---

- ▪ Le site officiel de Spring est <http://www.springframework.org>
- Les .jar sont disponibles à l'URL:  
<http://www.springsource.org/download>
  - version avec dépendances.
  - version avec librairies séparées.
- Spring version 2.5.6.SEC01 est la version courante.
  - nécessite la JDK 1.4+
- Spring version 2.0.8 est la dernière version 2.0.X
  - nécessite la JDK 1.3+
- La documentation officielle de Spring est disponible à l'URL  
<http://www.springsource.org/documentation>

# Les modules de Spring



# Les modules de Spring

---

- **Spring Core** : module de gestion des dépendances entre beans (implémente L'injection de dépendance).
- **Spring AOP** : réservé à des développements très spécifiques.
- **Spring ORM** : Classes utilitaires permettent d'intégrer les différentes framework de mapping O/R (IBATIS, Hibernate)
- **Spring DAO** : Classes utilitaires facilitent le développement d'une couche DAO en jdbc pur.
- **Spring Context** : permet de se connecter aux EJB, à JNDI, à JMS,...
- **Spring Web** : permet le développement des applications web.
- **Spring MVC** : Implémente une application web (basé sur le MVC)



# Design Pattern

---

- Singleton
  - Façade
  - IOC
  - AOP
  - Value Object

# Singleton

```
MonSingleton.java X
package singleton;

public class MonSingleton {

    private MonSingleton() {
        // TRAITEMENT COMMUNS EXECUTES UNE SEULE FOIS DOIT ETRE INSERER ICI
    }

    private static MonSingleton instance;

    public static MonSingleton getInstance() {
        if (instance == null)
            instance = new MonSingleton();
        return instance;
    }
}
```

➤ Voir TP 9

# Design Pattern

---

- Singleton
- **Façade**
- IOC
- AOP
- Value Object

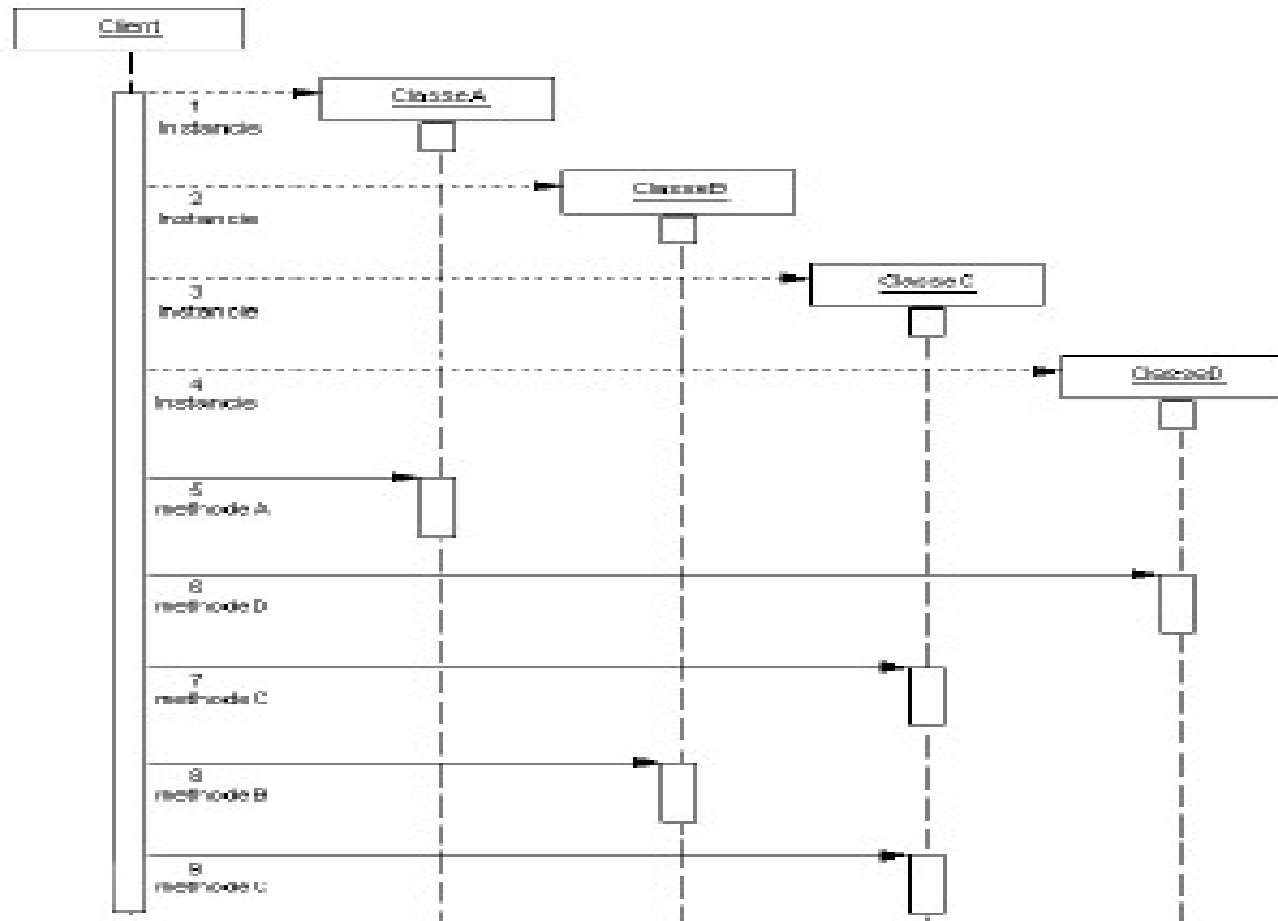
# Façade

---

- Une bonne pratique de conception est d'essayer de limiter le couplage existant entre des fonctionnalités proposées par différentes entités ;
- Dans la pratique, il est préférable de développer un petit nombre de classes et de proposer une classe pour les utiliser ;
- Le but est de proposer une interface facilitant la mise en œuvre d'un ensemble de classes généralement regroupées dans un ou plusieurs sous systèmes.

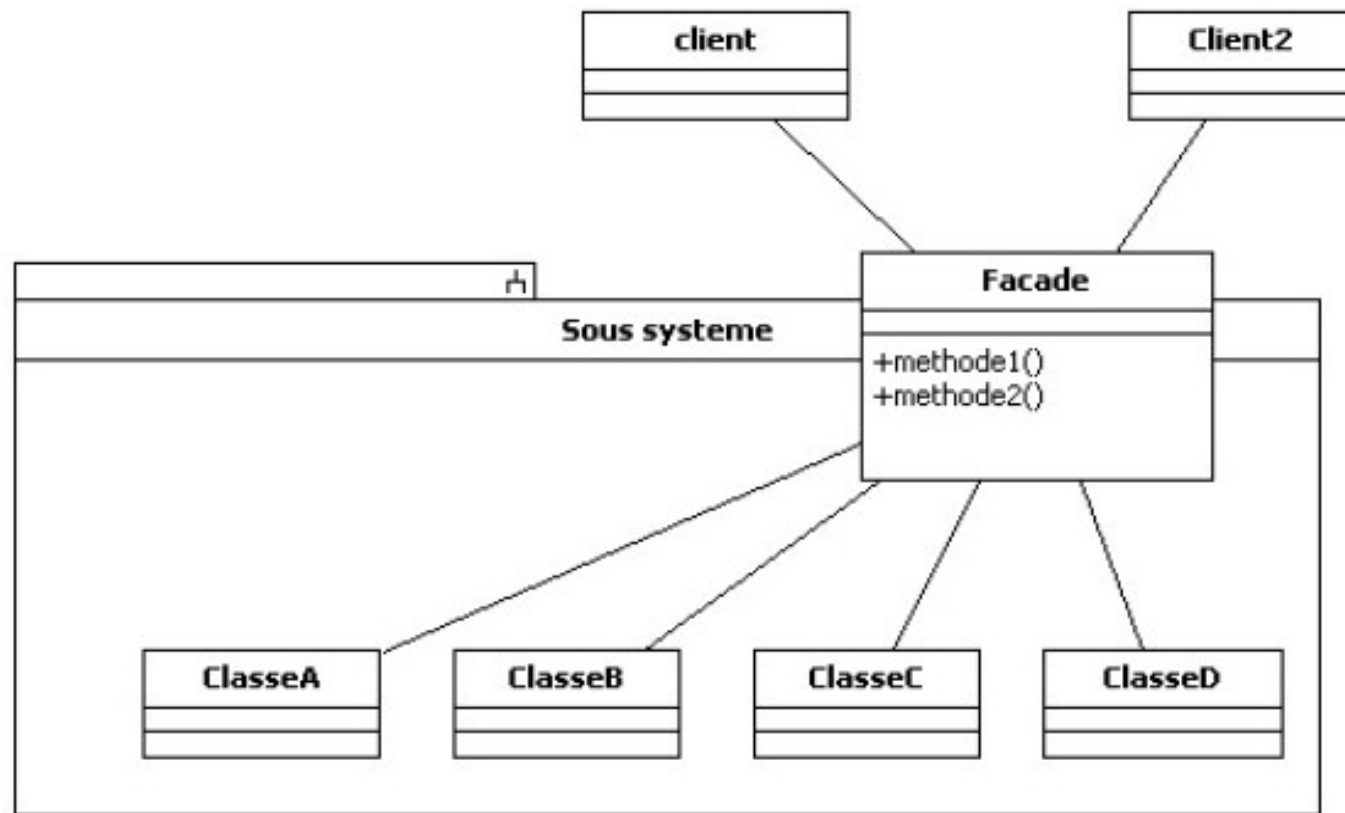
# Façade

## Exemple : Mauvaise pratique



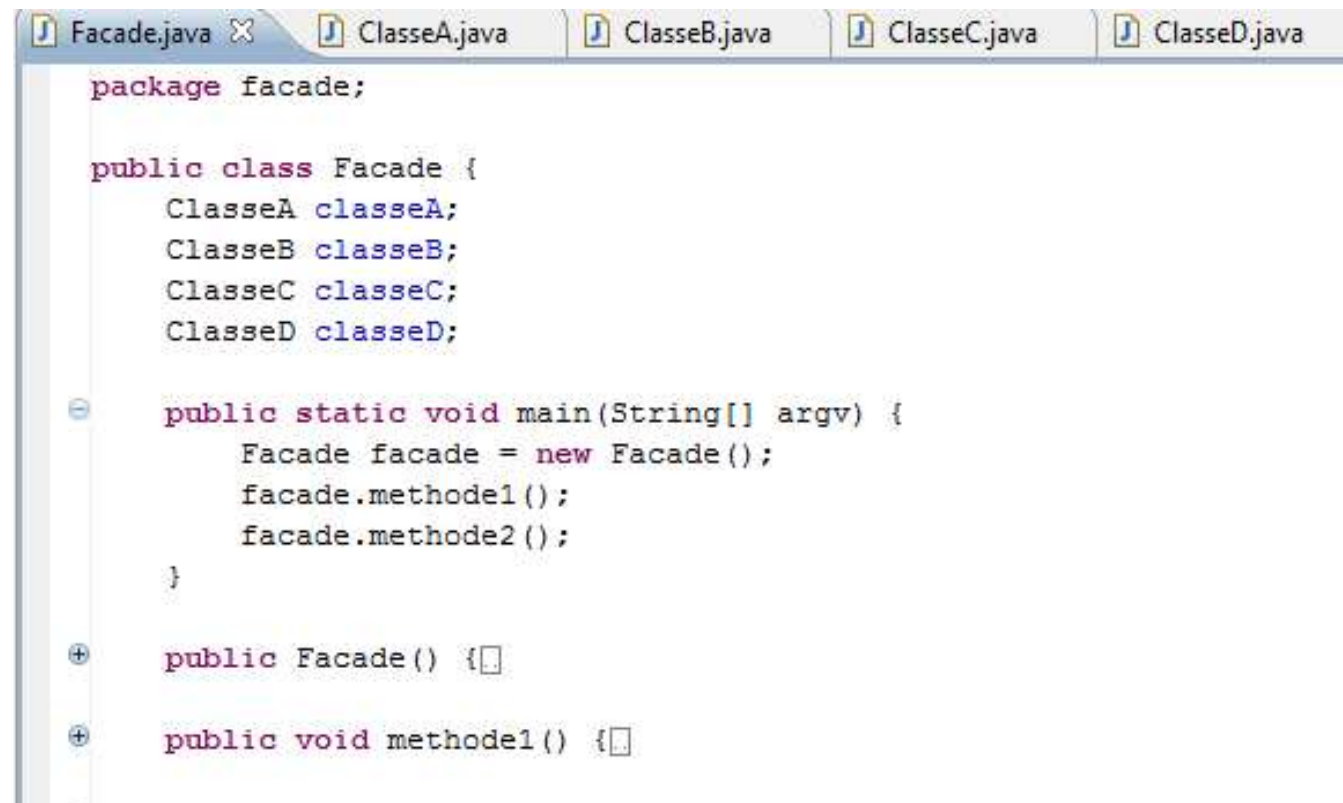
# Façade

## Exemple : Façade



# Façade

## ➤ Exemple : Façade



The screenshot shows a Java IDE with five tabs: Facade.java, ClasseA.java, ClasseB.java, ClasseC.java, and ClasseD.java. The Facade.java file is active and contains the following code:

```
package facade;

public class Facade {
    ClasseA classeA;
    ClasseB classeB;
    ClasseC classeC;
    ClasseD classeD;

    public static void main(String[] argv) {
        Facade facade = new Facade();
        facade.methode1();
        facade.methode2();
    }

    public Facade() {}

    public void methode1() {}
}
```

## ➤ Voit TP 10

# Design Pattern

---

- Singleton
  - Façade
- **IOC**
  - AOP
  - Value Object



# IOC (Spring Core)

---

## ■ Spring Core :

- est le noyau du Framework Spring
- est basé sur le Design Pattern IOC (Injection de Dépendances ou bien Inversion de Contrôle)
- se charge de l'instanciation de tous les objets de l'application et de la résolution des dépendances entre eux.

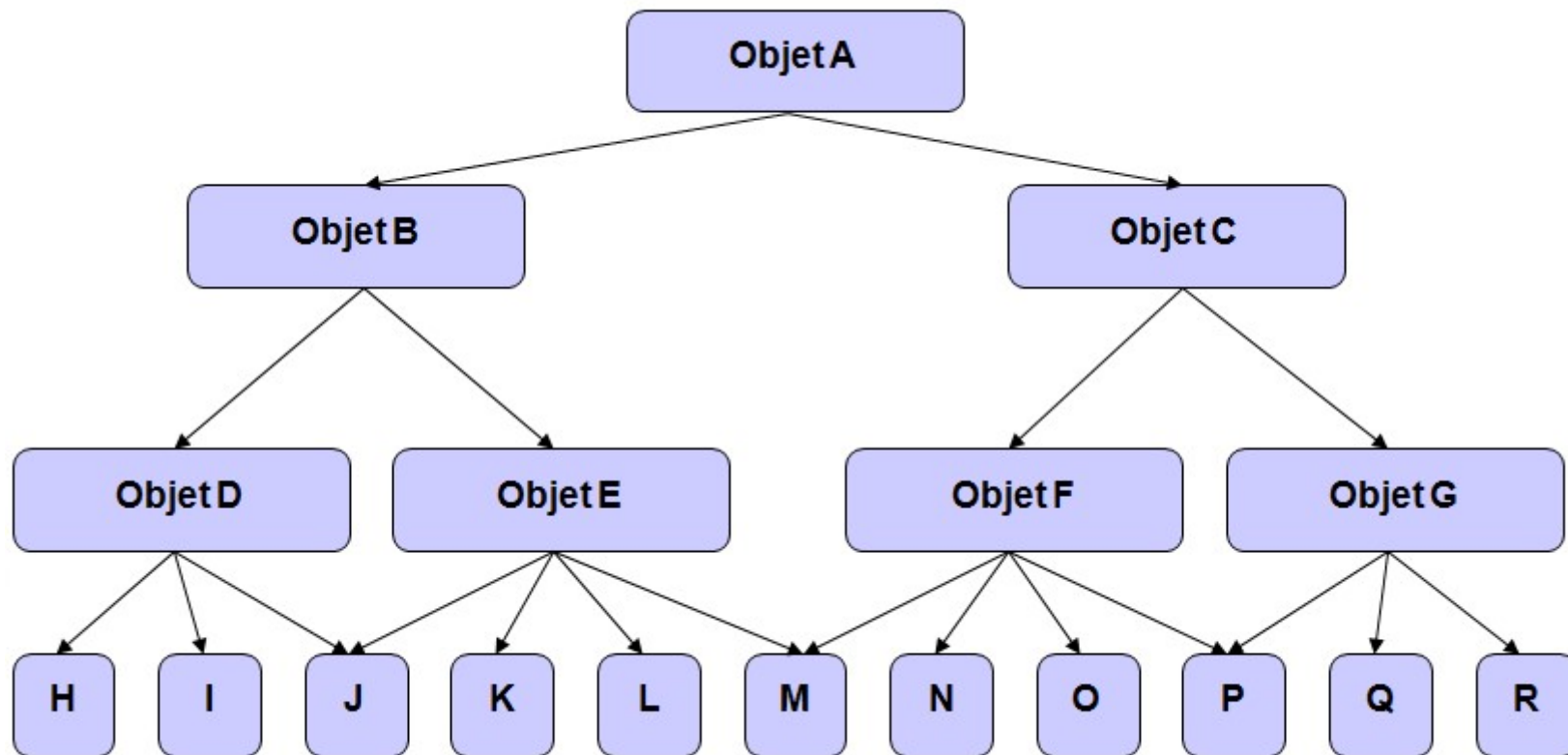
# Spring Core

---

- Deux API très importantes dans ce module :
  - `org.springframework.beans`
  - `org.springframework.context`
- Offrent les bases pour le Design Pattern IOC.
- BeanFactory (`org.springframework.beans.factory.BeanFactory`) permet de configurer les Beans (dans un fichier XML) et les gérer (instanciation, gestion de la dépendance).
- ApplicationContext (`org.springframework.context.ApplicationContext`) ajoute des fonctionnalités avancées au BeanFactory (facilite l'intégration avec Spring AOP, ..).

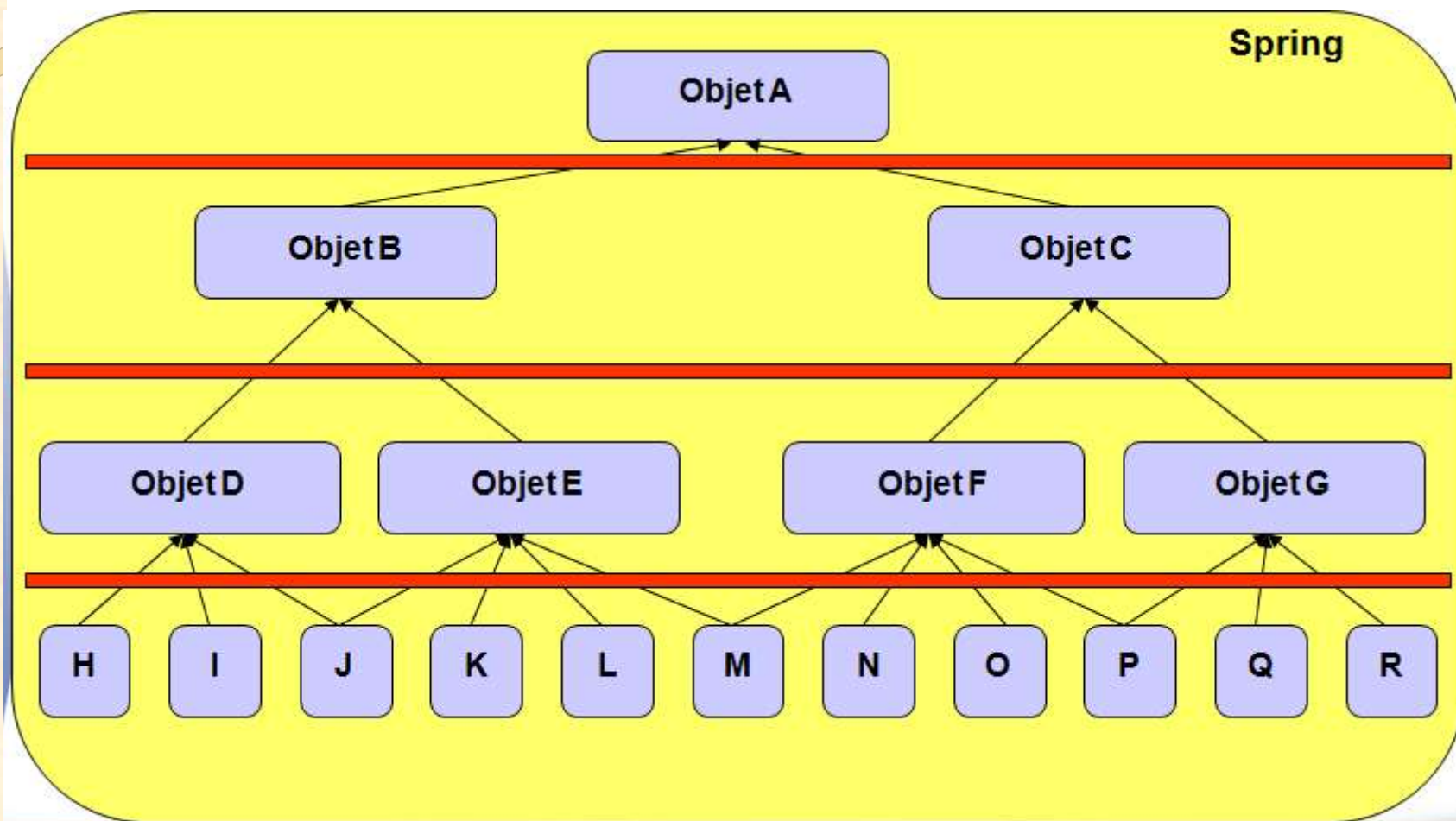
# Injection de dépendance

■ Approche classique :



# Injection de dépendance

- Injection de dépendance avec Spring :



# Injection de dépendance

---

- Spring va permettre à chaque couche de s'abstraire de sa ou ses couches inférieures (injection de dépendance) :
  - Le code de l'application est beaucoup plus lisible.
  - Le maintien de l'application est facilité.
  - Les tests unitaires sont simplifiés.
- Spring gère les dépendances entre les beans dans un fichier XML.

# Injection de dépendance

- Le format du fichier de configuration de Spring :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="personnel1" class="ma.emsi.formation.spring.Personne"
    init-method="init" destroy-method="close">
    <property name="nom">
      <value>Alami</value>
    </property>
    <property name="age">
      <value>40</value>
    </property>
  </bean>
  <bean id="personne2" class="ma.emsi.formation.spring.Personne"
    init-method="init" destroy-method="close">
    <property name="nom">
      <value>Amrani</value>
    </property>
    <property name="age">
      <value>20</value>
    </property>
  </bean>
</beans>
```

# Fabrique de Bean et contexte d'application

---

- Fabrique de Bean
  - Le contexte d'application
  - Définition d'un Bean
  - Les méthodes d'injection

# Fabrique de Bean

---

- La fabrique de Bean (Bean Factory) est l'interface de base permettant aux applications reposant sur le conteneur léger de Spring d'accéder à ce dernier.
- Définit les fonctionnalités minimales dont dispose l'application pour dialoguer avec le conteneur.
- Cette interface comporte plusieurs implémentations prêtes à l'emploi.
- Il est possible de définir notre propre implémentation (Spring est un framework OUVERT)



# Fabrique de Bean

---

- Une application accède aux Beans du Spring moyennant les deux interfaces suivantes :
  - L'interface BeanFactory
    - définit l'accès aux Beans gérés par Spring
  - BeanDefinitionRegistry
    - formalise la définition des Beans que Spring doit gérer
- Ces interfaces disposent plusieurs implémentations.
- Peuvent aussi être implémentées spécifiquement dans le cadre d'un projet

# Fabrique de Bean

---

Public interface BeanFactory {

Object getBean(String name) throws BeansException;

Object getBean(String name, Class requiredType) throws BeansException;

boolean containsBean(String name);

boolean isSingleton(String arg0) throws NoSuchBeanDefinitionException;

Class getType(String name) throws NoSuchBeanDefinitionException;

String[] getAliases(String arg0) throws NoSuchBeanDefinitionException;  
}

# Fabrique de Bean

---

- BeanFactory est étendue par d'autres interfaces :
  - ConfigurableBeanFactory
  - ListableBeanFactory
  - HierarchicalBeanFactory
- BeanDefinitionRegistry
  - Spécifie le protocole pour définir les objets à gérer par Spring ainsi que leurs dépendances

# Fabrique de Bean

```
Public interface BeanDefinitionRegistry {  
  
    int getBeanDefinitionCount();  
  
    String[] getBeanDefinitionNames();  
  
    boolean containsBeanDefinition(String arg0);  
  
    BeanDefinition getBeanDefinition(String arg0) throws NoSuchBeanDefinitionException;  
  
    void registerBeanDefinition(String arg0, BeanDefinition arg1) throws BeansException;  
  
    String[] getAliases(String arg0) throws NoSuchBeanDefinitionException;  
  
    void registerAlias(String arg0, String arg1) throws BeansException;  
}
```

# Fabrique de Bean

---

- Implémentations de BeanFactory et BeanDefinitionRegistry :
  - DefaultListableBeanFactory
    - `org.springframework.beans.factory.support`
  - XmlBeanFactory
    - supporte la définition des objets sous format XML
      - son fichier XSD: [www.springframework.org/schema/beans](http://www.springframework.org/schema/beans)
- La définition des objets sous forme XML est la plus utilisée dans les projets de développement basés sur Spring.

# Fabrique de Bean et contexte d'application

---

- Fabrique de Bean
  - Le contexte d'application
  - Définition d'un Bean
  - Les méthodes d'injection

# Le contexte d'application

---

- En plus des fonctionnalités offertes par la fabrique et le registre de Bean :
  - support des messages et de leur internationalisation
  - Support avancé de chargement de fichiers (appelés ressources)
  - Support de la publication d'événements, permettant à des objets de l'application de réagir en fonction d'eux
  - Possibilité de définir une hiérarchie de contextes. Cette fonctionnalité est très utile pour isoler les différentes couches de l'application (les Beans de la couche présentation ne sont pas visibles de la couche service, par exemple)

# Le contexte d'application

Spring offre plusieurs implémentations de l'interface `ApplicationContext` :

- `FileSystemXmlApplicationContext`
- `ClassPathXmlApplicationContext`

▪ Exemple:

```
public static void main(String[] args) {  
    ApplicationContext context=new FileSystemXmlApplicationContext("applicationContext.xml");  
    UnBean monBean=(UnBean) context.getBean("monBean");  
}
```



# Le contexte d'application

---

- Par convention, les noms des fichiers de définition des objets s'appellent *applicationContext.xml*
  - Si la définition des objets se fait dans plusieurs fichiers XML, alors les noms des fichiers sont préfixés par *applicationContext-* et conservent l'extension *.xml*.
  - Le respect de cette norme n'est pas obligatoire.
- Exemple:
  - *applicationContext.xml*
  - *applicationContext-hibernate.xml*
  - *applicationContext-security.xml*

# Fabrique de Bean et contexte d'application

---

- Fabrique de Bean
  - Le contexte d'application
  - Définition d'un Bean
  - Les méthodes d'injection

# Définition d'un Bean

---

## ■ Définition d'un Bean :

- Dans Spring, la notion de Bean correspond à celle d'instance de classe.
- Cette classe peut être un `JavaBean` ou bien une classe quelconque
- pour que Spring initialise correctement le Bean, il faut définir le constructeur ou bien les modificateurs.
- Spring utilise aussi des fabrique spécifiques (Factory) pour les instantiations complexes.
- La définition d'un Bean peut s'effectuer de manière programmatique :
  - via la méthode `registerBeanDefinition` de l'interface `BeanDefinitionRegistry`
  - un fichier XML (Recommandé)

# Définition d'un Bean

---

- Les informations de base

- La définition d'un Bean nécessite au minimum la fourniture de son type (sa classe) et son nom
- Spécifier à Spring si le Bean est un Singleton ou un Prototype

- Structure du fichier de configuration

- la définition des Beans s'effectue dans un fichier XML
- Spring utilise pour ceci des schémas spécifiques
- la racine du fichier XML est *beans*
- il existe des schémas pour la définition des Beans, pour AOP (Aspect Oriented Application), pour la gestion des transactions et autres ...

# Définition d'un Bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd"
</beans>
```

# Définition d'un Bean

## ■ Nommage des Beans :

- Un Bean est défini par le tag *bean* et identifié par au moins deux informations: *class* et *name*

- Exemple: `<bean id="personne1" class="monPackage.Personne"/>`

Le résultat de cette configuration est une instance *personne1* dont les propriétés ne sont pas encore initialisées.

- *id* est un identifiant unique

- Pour définir des alias à un Bean donné, Spring utilise l'attribut *name*.  
Voici la syntaxe :

`<bean id="personne1" class="monPackage.Personne" name="alias1,alias2,alias3"/>`

Spring accepte aussi les espaces à la place de « , »

## ■ Sélection du mode d'instanciation :

- par défaut, Spring considère que les Beans sont des singletons: si nous appelons plusieurs fois la méthode *getBean* du contexte d'application, nous obtenons donc toujours le même objet.

la syntaxe :

`<bean id="personne1" class="monPackage.Personne" singleton="true" />`

# Fabrique de Bean et contexte d'application

---

- Fabrique de Bean
  - Le contexte d'application
  - Définition d'un Bean
  - Les méthodes d'injection

# Les méthodes d'injection

---

- ▪ Injection par modificateur
- Injection par constructeur
- Injection des propriétés
  - Injection de valeurs simples
  - Injection de la valeur null
  - Injection de structures de données
  - Injection des collaborateurs
- Injection par Factory



# Injection par modificateur



```
<bean id="mysqlDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/tudu?characterEncoding=UTF-8" />
  <property name="username" value="root" />
  <property name="password" value="" />
  <property name="maxActive" value="50" />
  <property name="maxIdle" value="30" />
  <property name="maxWait" value="1000" />
  <property name="poolPreparedStatements" value="true" />
  <property name="maxOpenPreparedStatements" value="-1" />
</bean>
```

# Injection par modificateur

- Un modificateur ne correspond pas forcément à un attribut de l'objet à initialiser mais il peut s'agir d'un traitement d'initialisation plus complexe.
- Le tag *property* s'utilise en combinaison avec le tag *value* qui sert à spécifier la valeur à affecter à la propriété lorsqu'il s'agit d'une propriété canonique, ou avec le tag *ref*, s'il s'agit d'un *collaborateur*.
- la syntaxe suivante est valide :

```
<bean id="bean I" class="class I">  
  <property name="attribut I" value="valeur I"/>  
</bean>
```

# Injection par constructeur

```
<bean id="voiture1" class="ma.emsi.formation.spring.Voiture"
      init-method="init" destroy-method="close">
  <constructor-arg index="0">
    <value>Peugeot</value>
  </constructor-arg>
  <constructor-arg index="1">
    <value>307</value>
  </constructor-arg>
  <constructor-arg index="2">
    <ref bean="personne2"></ref>
  </constructor-arg>
</bean>
```

# Injection par constructeur

- L'injection par constructeur se paramètre au sein d'une définition de Bean en spécifiant les paramètres du constructeur par le biais du tag *constructor-arg*

- Il est possible de changer l'ordre de définition des paramètres du constructeur en utilisant le paramètre *index* du tag *constructor-arg*

- L'indexation se fait à partir de 0

```
<bean id="monbean" class="maclasse">  
  <constructor-arg index="1"  
    <value>10</value>  
  </constructor-arg>  
  <constructor-arg index="0"  
    <value>valeur</value>  
  </constructor-arg>  
</bean>
```

# Injection par constructeur

- Par défaut, Spring utilise le premier constructeur supportant la définition du Bean dans le document XML.

## ■ Exemple :

```
public class UnBean {  
  
    public UnBean(String chaine) {  
        this.chaine=chaine;  
        this.entier=0;  
    }  
    public UnBean(int entier) {  
        this.chaine="";  
        this.entier=entier;  
    }  
}
```

```
<bean id="monBean" class="UnBean">  
    <constructor-arg>  
        <value>10</value>  
    </constructor-arg>  
</bean>
```

Dans ce cas Spring utilise le premier constructeur.

# Injection par constructeur

- Pour lever cette ambiguïté, il est possible d'utiliser le paramètre *type* du tag *constructor-arg* :

```
<bean id="monBean" class="UnBean">  
  <constructor-arg type="int">  
    <value>10</value>  
  </constructor-arg>  
</bean>
```

```
<bean id="monBean" class="UnBean">  
  <constructor-arg type="java.lang.String">  
    <value>10</value>  
  </constructor-arg>  
</bean>
```

# Injection par constructeur

---

- C'est le type de l'argument est une classe, Spring recommande d'utiliser son package complet.
- La syntaxe suivante est valide :

```
<bean id="monBean" class="UnBean">  
  <constructor-arg type="java.lang.String" value="valeur l"/>  
</bean>
```

# Injection des propriétés

---

- L'injection des propriétés est le mécanisme fondamental du conteneur léger.
- Spring supporte l'injection de valeurs simples en convertissant les chaînes de caractères fournies au tag ou au paramètre *value* dans le type de la propriété à initialiser.
- Les types de propriétés supportés par Spring sont :
  - Booléens;
  - Type char et java.lang.Character
  - Type java.util.Locale
  - Type java.net.URL
  - Type java.io.File
  - Type java.lang.Class
  - Tableaux de bytes
  - Tableaux de chaînes de caractères



# Injection des propriétés

```
public class UnBean {  
    private String chaine;  
    private int entier;  
    private float reel;  
    private boolean booleen;  
    private char caractere;  
    private java.util.Properties properties;  
    private java.util.Locale localisation;  
    private java.net.URL url;  
    private java.io.File fichier;  
    private java.lang.Class classe;  
    private byte[] tab2bytes;  
    private String[] tab2chaines;  
}
```

# Injection des propriétés

```
<bean id="monBean" class="UnBean">  
  <property name="chaine" valeur="valeur"/>  
  <property name="entier" valeur="10"/>  
  <property name="reel" valeur="10.5"/>  
  <property name="booleen" valeur="true"/>  
  <property name="caractere" valeur="a"/>  
  <property name="properties"  
    valeur="log4j.rootLogger=DEBUG.CONSOLE\nlog4j.logger.appli=WARN"  
  />  
  <property name="localisation" valeur="fr_FR"/>  
  <property name="url" valeur="http://appli.ma"/>  
  <property name="fichier" valeur="file:c:\\temp\\test.txt"/>  
  <property name="classe" valeur="java.lang.String"/>  
  <property name="tab2bytes" valeur="valeur"/>  
  <property name="tab2chaines" valeur="valeur1,valeur2"/>  
</bean>
```

# Injection des propriétés

- Pour les attributs *properties* et *localisation*, le format de la chaîne de caractères respecte le format attendu pour ces types de données (ce format est spécifié dans l'API J2SE)
- Pour la propriété fichier, le format utilisé est celui des URL. Ce format support le protocole classpath introduit par Spring pour accéder aux fichiers se trouvant dans le classpath.
- Injection de la valeur null :

```
<bean id="monBean" class="UnBean">  
  <constructor-arg type="java.lang.String">  
    <null/>  
  </constructor-arg >  
</bean>
```

# Injection des propriétés

- Injection de structure de données:

- Spring supporte les structures suivantes :

- java.util.Map
- java.util.Set
- java.util.List

- Le type java.util.Map :

```
<property name="maMap">
  <map>
    <entry key="cle1">
      <value>valeur1</value>
    </entry>
    <entry key="cle2">
      <value>valeur2</value>
    </entry>
  </map>
```

# Injection des propriétés

- Le type `java.util.Map` :

- La syntaxe suivante est valide aussi :

```
<property name="maMap">  
  <map>  
    <entry key="cle1" value="value1"/>  
    <entry key="cle2" value="value2"/>  
  </map>
```

- Le type `java.util.Set` :

```
<property name="monSet">  
  <set>  
    <value>valeur1</value>  
    <value>valeur2</value>  
  </set>  
</property>
```

# Injection des propriétés

- Le type `java.util.List` :

```
<property name="maListe">
  <list>
    <value>valeur 1 </value>
    <value>valeur 1 </value>
  </list>
</property>
```

- Le type `java.util.Properties` :

```
<property name="properties">
  <props>
    <prop key="log4j.rootLogger">
      DEBUG.CONSOLE
    </prop>
    <prop key="log4j.logger.appli">
      WARN
    </prop>
  </props>
</property>
```

# Injection des propriétés

## ■ Injection des collaborateurs :

- Spring utilise deux méthodes pour l'injection des collaborateurs :
  - Injection explicite des collaborateurs
  - Injection automatique

## ■ Injection explicite :

- c'est le développeur qui décide du choix du Bean.

```
<bean id="serviceManager" class="ServiceManagerImpl">  
  <property name="dao">  
    <ref bean="dao"/>  
  </property>  
</bean>
```

```
<bean id="serviceManager" class="ServiceManagerImpl">  
  <property name="dao">  
    <ref local="dao"/>  
  </property>  
</bean>
```

# Injection des propriétés

## ▪ Injection explicite, autre syntaxe :

```
<bean id="serviceManger" class="ServiceManagerImpl">
  <property name="dao">
    <bean class="DaoManagerImpl">
      <property name="attr1" value="valeur1"/>
      <property name="attr2" value="valeur2"/>
    </bean>
  </property>
</bean>
```

➔ Le Bean DaoManagerImpl n'est visible qu'à l'intérieur du Bean serviceManger



# Injection des propriétés

## ♀ Injection automatiques des collaborateurs :

- sur des projets de grande taille, les configurations peuvent rapidement devenir imposantes !
  - Spring propose un mécanisme d'injection automatique, appelé *autowiring*
  - ce mécanisme utilise des algorithmes de décision pour savoir quelle injection à réaliser
  - L'autowiring est activé Bean par Bean par le biais du paramètre autowire du tag bean
  - Par défaut, l'autowiring n'est pas activée
- Spring propose 4 types d'algorithmes :
  - byName: Spring cherche le Bean ayant le même nom de la property
  - byType : Spring cherche le Bean ayant le même type de la property
  - constructor : fondé sur les paramètres du constructeur
  - autodetect : sélectionne automatiquement la recherche par le type ou par le constructeur.

# Injection des propriétés

## ■ Injection automatique: byName

```
<bean id="personnes" class="ma.emsi.formation.spring.Personnes"
      autowire="byName" />

<bean id="personnel1" class="ma.emsi.formation.spring.Personne">
  <property name="nom" value="Alami" />
  <property name="age" value="40" />
</bean>
<bean id="personne2" class="ma.emsi.formation.spring.Personne">
  <property name="nom" value="Alami" />
  <property name="age" value="40" />
</bean>
```

```
package ma.emsi.formation.spring;

public class Personnes {
    Personne personnel1;
    Personne personne2;
}
```

# Injection des propriétés

## ■ Injection automatique: byType

```
<bean id="personnes" class="ma.emsi.formation.spring.Personnes"
      autowire="byType" />

<bean id="personne2" class="ma.emsi.formation.spring.Personne">
  <property name="nom" value="Alami" />
  <property name="age" value="40" />
</bean>
```

```
package ma.emsi.formation.spring;

public class Personnes {
    Personne personne4;
    Personne personne5;
}
```

# Injection des propriétés

## ■ Injection automatique: constructor

```
<bean id="personnes" class="ma.emsi.formation.spring.Personnes"
      autowire="constructor" />

<bean id="personne2" class="ma.emsi.formation.spring.Personne">
  <property name="nom" value="Alami" />
  <property name="age" value="40" />
</bean>
```

```
package ma.emsi.formation.spring;

public class Personnes {
    Personne personne4;
    Personne personne5;


    public Personnes(Personne personne4, Personne personne5) {
        this.personne4 = personne4;
        this.personne5 = personne5;
    }
}
```

## ■ Voir TP 10

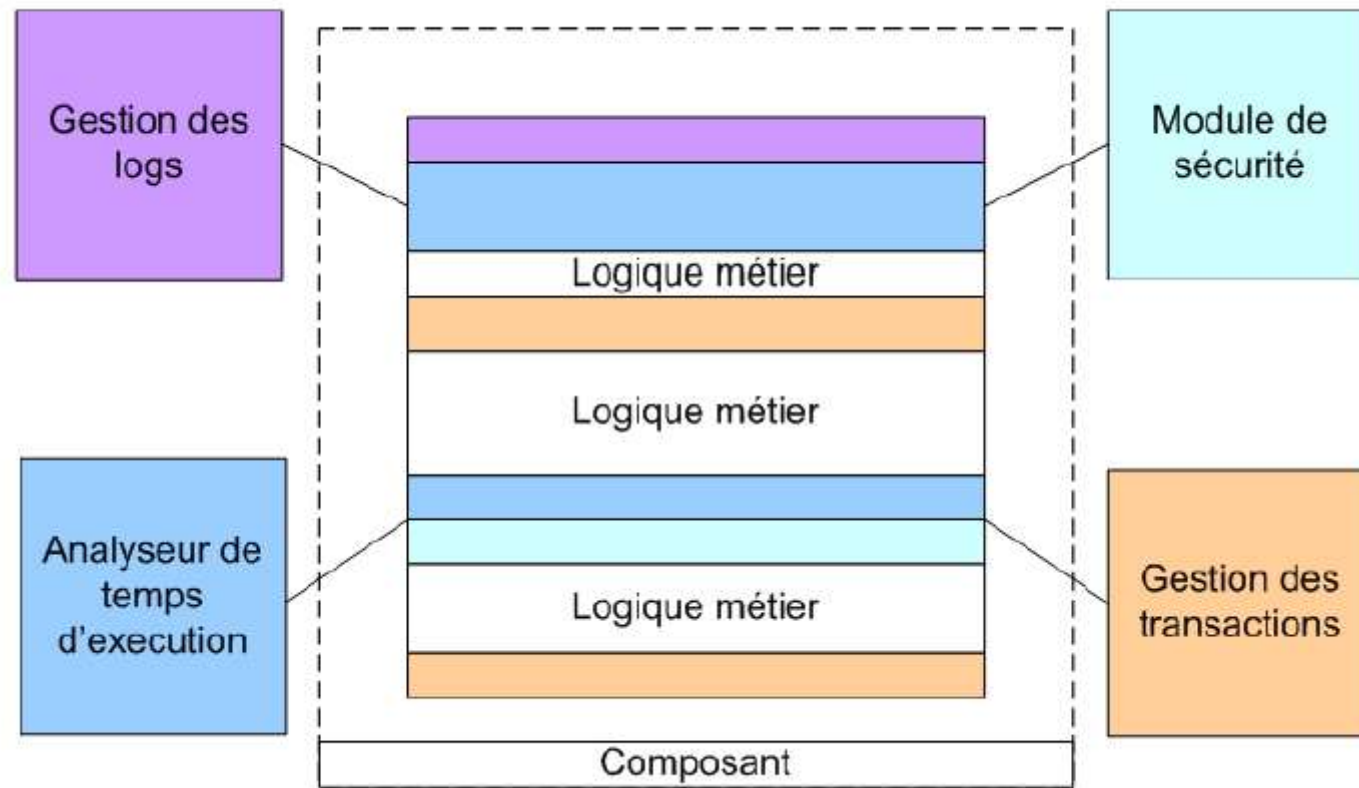
Composants d'entreprise

# Design Pattern

---

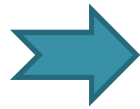
- 
- Singleton
    - Façade
  - IOC
    - AOP
    - Value Object

# AOP (Spring AOP)



# Spring AOP

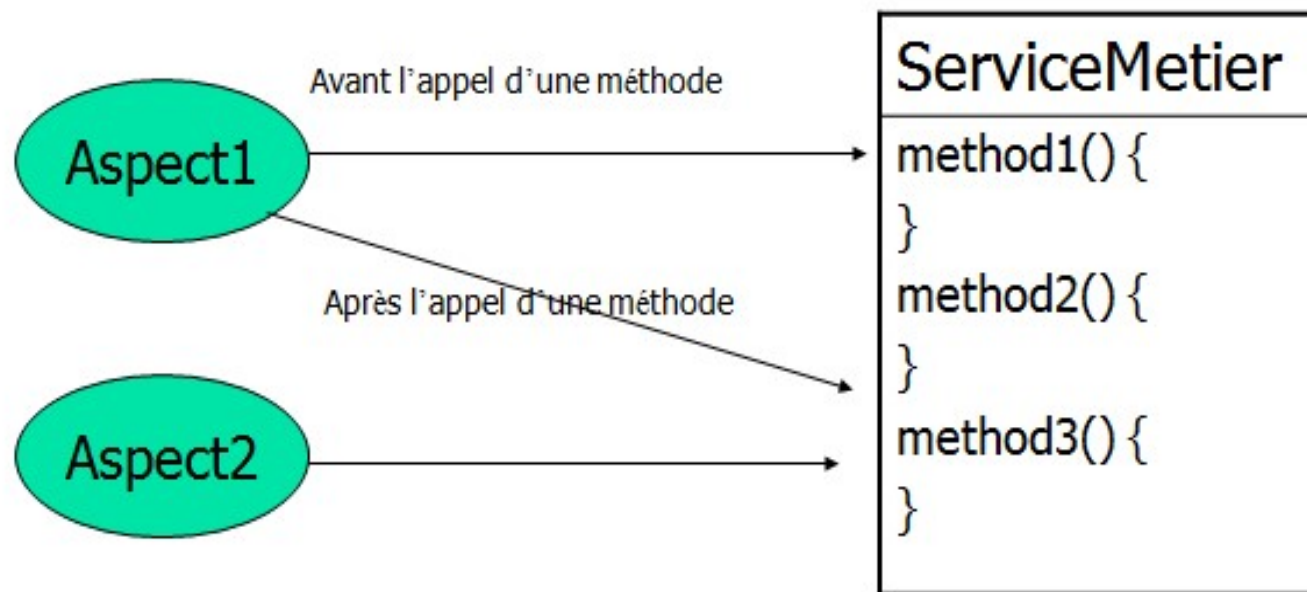
- Un module ou composant métier est régulièrement pollué par de multiples appels à des composants utilitaires externes.



- Code pollué des traitements techniques
- Faible réutilisabilité
- Qualité plus basse due à la complexité du code
- Difficulté à faire évoluer

- La solution constituerait donc à externaliser tous les traitements non relatifs à la logique métier en dehors du composant.
- Il faut pouvoir définir des traitements de façon déclarative ou programmés dans des points clés de l'algorithme (avant ou après une méthode)

# Spring AOP





# Spring AOP

---

- La démarche est alors la suivante :
  - Séparer la partie métier de la partie utilitaire.
  - Se concentrer sur la partie métier
  - Définition des aspects

# Spring AOP

---

- Il existe plusieurs solutions implémentant l'AOP :
  - Xerox dans les années 90.
  - AspectJ : extension pour Java, devenu projet de la fondation Eclipse.
  - D'autres solutions :
    - Java : Spring AOP, JBoss AOP, AspectWerkz,...
    - C/C++ ;
    - PHP ;
    - .NET.

# De l'objet à l'Aspect: Notions de base

---

- Joinpoint : un point précis du code où l'aspect sera exécuté
  - Début d'une méthode
  - Fin d'une méthode
  - Si l'appel d'une méthode provient d'une autre ...
- Pointcut : une collection d'une ou plusieurs Joinpoint : par nom de méthode, par nom de classe, ...
- Advice : le code qui sera exécuté, il y a différents types d'advice :
  - Before : exécution du code avant le pointcut
  - After : exécution du code après le pointcut
  - Around : exécution du code avant et après le pointcut
- L'aspect : combinaison entre un advice et un pointcut

## De l'objet à l'Aspect: Notions de base

- Exemples de pointcut : `say*`, `sayHello*`, `sayHelloBefore`, `say(String,int)`
- Exemple d'advice : afficher un message avant `sayHelloBefore`
- Exemple d'aspect : `sayHelloBefore`, afficher un message

### **MonAspectExemple**

```
void sayHelloBefore()  
void sayHelloAfter()  
void sayHelloAround()  
void say(String what2say, int nbrFois)
```

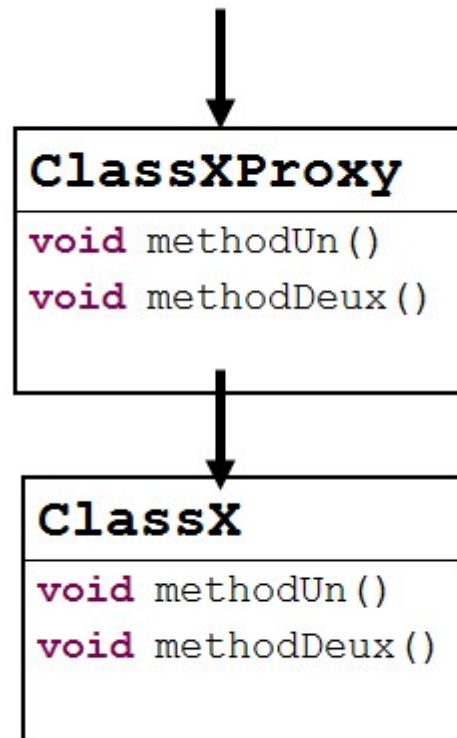
# Spring AOP

---

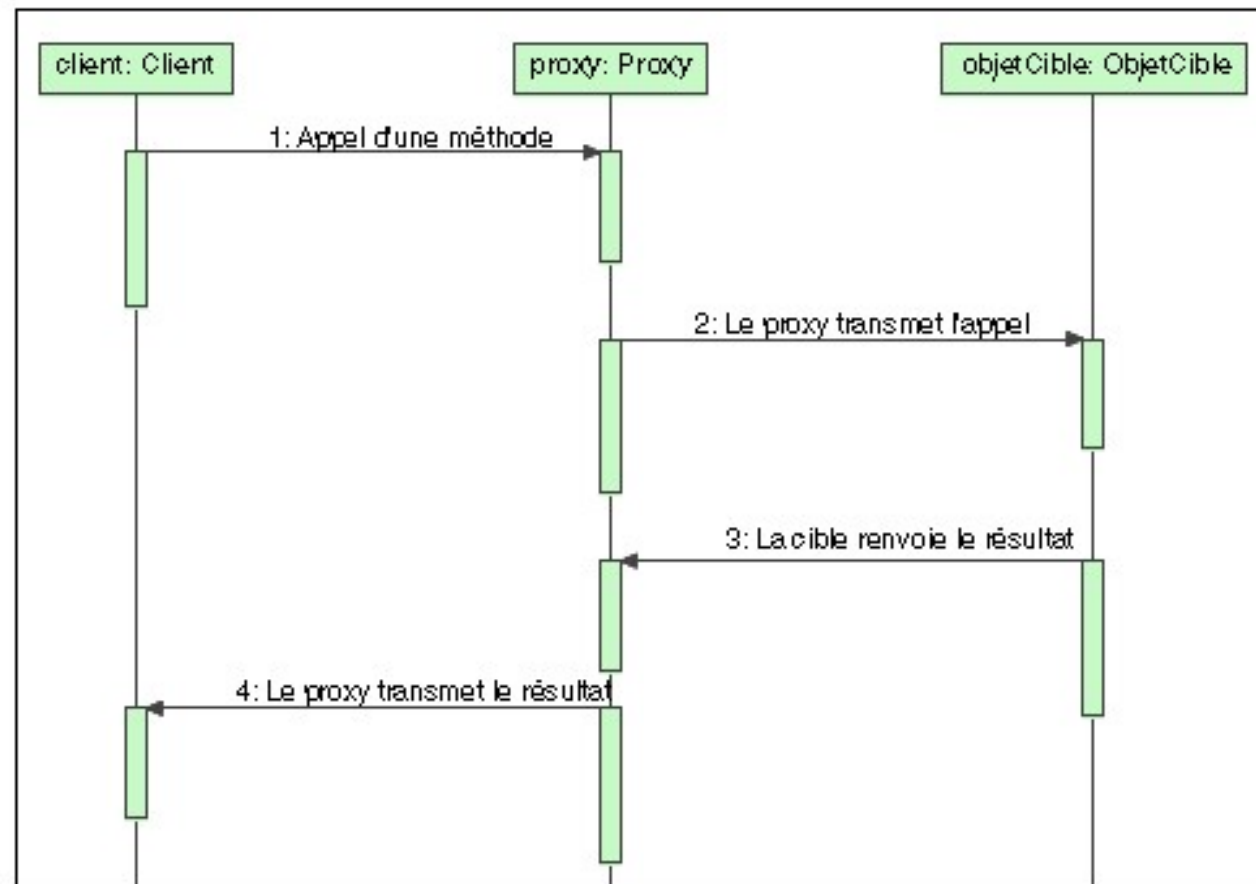
- Comment on procède pour appliquer les aspects ?
  - Méthode statique : enrichissement des bytecode, fait au moment de la compilation (Exemple de AspectJ)
  - Méthode dynamique : génération de proxies pour intercepter les appels vers les méthodes (Exemple de Spring AOP)
- Spring permet l'intégration d'AspectJ

# Spring AOP

---



# Spring AOP: principe d'un proxy



# Spring AOP

---

- ■ Des exemples en AspectJ :
  - Plugin AJDT pour AspectJ sous Eclipse
  - AspectJ utilise la méthode statique pour appliquer les aspects
  - AspectJ dispose d'un langage spécifique pour les aspects (qui ressemble à Java)
- Les mêmes exemples en Spring AOP :
  - Utilise les proxies pour appliquer les aspects
  - Applique les aspects par déclaration ou par programmation



# Spring AOP

---

## Les transactions :

- Exécuter un groupe de requête : tout le groupe ou rien
- Spring fournit un Proxy tout prêt pour cela :  
*TransactionProxyFactoryBean*
- L'aspect transactionnel est déclaré dans un fichier XML
- On précise les propriétés de la transaction dans un fichier XML

# Spring AOP

---


Avec Spring, il est possible de tracer les appels aux méthodes objets pendant l'exécution ET SANS modifier le code de l'application.

(voir TP 10)

s

# Design Pattern

---

- 
- Singleton
    - Façade
  - IOC
    - AOP
    - Value Object

# Value Object

---



➤ Voir TP 12