

UNIVERSITE CADY AYYAD
ECOLE SUPERIEURE DE TECHNOLOGIE
ESSAOUIRA



L'Essentiel du langage java avancé

Pour la Licence Informatique

ISIL

- 2014/2019 -

Sommaire

I.	Introduction de Java Swing.....	4
II.	Outils de developpement de Java.....	6
	1. L'installation de JDK ou JRE	6
	2. Eclipse IDE	7
	3. Lancement d'eclipse IDE.....	7
III.	Les composantes Java Swing.....	8
	1. Les classes mères component et JComponent	8
	2. La relation entre les composantes	9
	3. Les propriétés géométriques	9
	4 . Les composantes graphiques.....	11
IV.	Le Modèle MVC.....	16
	1. L'architecture MVC	16
	2. Les avantages et Inconvénients de MVC	17
V.	Les Thread	18
	1. Introduction	18
	1.1. Rappel sur les processus	18
	1.2. Les Threads	19
	2. Cycle de vie d'un Thread	20
	3. Création d'un Thread	21
	4. Sommeil d'un Thread	22
	5. Synchronisation	23
	6 . Coordination entre les Threads	24
VI.	Les annotations & la réflexion.....	25
	1. Introduction sur les annotations	25
	2 . Qu'est-ce qu'une Annotation ?	25
	3. Création d'une annotation	29
	4 . Introduction sur la réflexion.....	31
	5. Mise en œuvre de la réflexion	31
	6. Réflexion et sécurité	37

VII.	La spécification JPA	38
1.	ORM (Object Relational Mapping)	38
2.	Les Entités.....	39
3.	Les associations (mapping)	42
4.	Le comportement	48
5.	Les Requêtes	51
6.	Exemples de mise en œuvre.....	57

I-Introduction de Java Swing

Swing est une bibliothèque graphique pour le langage de programmation Java, faisant partie du package Java Foundation Classes(JFC), inclus dans J2SE. Swing constitue l'une des principales évolutions apportées par Java 2 par rapport aux versions antérieures.

Il utilise le principe Modèle-Vue-Contrôleur (MVC, les composants Swing jouent en fait le rôle du contrôleur au sens du MVC) et dispose de plusieurs choix d'apparence (de vue) pour chacun des composants standards.

Le MVC (Model View Controller), Modèle, Vue, Contrôleur comprend :

- Un modèle. Celui-ci comprend les données et un certain nombre de méthodes pour les lire et les modifier. Le modèle peut toutefois enregistrer une ou plusieurs vues qui sont notifiées quand les données du modèle ont subi une modification (patron de conception Observateur).
- Une ou plusieurs vues. Une vue fournit une représentation graphique de tout ou partie des données du modèle. Les vues sont construites à base de composants Swing. Les vues doivent cependant implémenter l'interface `java.util.Observer`.
- Un contrôleur : son rôle est de traiter les événements en provenance de l'interface utilisateur et les transmet au modèle pour le faire évoluer ou à la vue pour modifier son aspect visuel.

L'objectif de ce cours est de présenter le design pattern, modèle de conception, Modèle Vue Contrôleur (MVC). Pour cela, ce cours fera une description générale du MVC. En effet, le Design pattern aide à avoir une meilleure manière pour analyser un problème. Le modèle Model View Controller a pour objectif d'améliorer le style et l'efficacité le plus. Il a conduit à de nouvelles façons d'écrire des applications entières, en améliorant constamment les applications . Ensuite ce cours illustre progressivement les composantes de MVC et leurs interactions . La dernière partie est consacrée à une implémentation en JAVA de ce design pattern.

Swing fait partie de la bibliothèque Java Foundation Classes (JFC). C'est une API dont le but est similaire à celui de l'API AWT mais dont les modes de fonctionnement et

d'utilisation sont complètement différents. Swing a été intégré au JDK depuis sa version 1.2. Cette bibliothèque existe séparément. pour le JDK 1.1. Tous les éléments de Swing font partie d'un package qui a changé plusieurs fois de nom : le nom du package dépend de la version du J.D.K. utilisée :

- `com.sun.java.swing` : jusqu'à la version 1.1 beta 2 de Swing, de la version 1.1 des JFC et de la version 1.2 beta 4 du J.D.K.
- `java.awt.swing` : utilisé par le J.D.K. 1.2 beta 2 et 3
- `javax.swing` : à partir des versions de Swing 1.1 beta 3 et J.D.K. 1.2 RC1

Les composants Swing forment une nouvelle hiérarchie parallèle à celle de l'AWT. L'ancêtre de cette hiérarchie est le composant `JComponent`. Presque tous ses composants sont écrits en pur Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : `JApplet`, `JDialog`, `JFrame`, et `JWindow`. Cela permet aux composants de toujours avoir la même apparence quelque soit le système sur lequel l'application s'exécute. Tous les composants Swing possèdent les caractéristiques suivantes :

- ce sont des beans
- ce sont des composants légers (pas de partie native) hormis quelques exceptions.
- leurs bords peuvent être changés

La procédure à suivre pour utiliser un composant Swing est identique à celle des composants de la bibliothèque AWT : créer le composant en appelant son constructeur, appeler les méthodes du composant si nécessaire pour le personnaliser et l'ajouter dans un conteneur.

Swing utilise la même infrastructure de classes qu'AWT, ce qui permet de mélanger des composants Swing et AWT dans la même interface. Il est toutefois recommandé d'éviter de les utiliser simultanément car certains peuvent ne pas être restitués correctement. Les composants Swing utilisent des modèles pour contenir leurs états ou leurs données. Ces modèles sont des classes particulières qui possèdent toutes un comportement par défaut.

II-Outils de developpement de Java

1. L'installation de JDK ou JRE:

Téléchargez l'environnement Java sur le site d'Oracle, comme le montre la figure suivante. Choisissez la dernière version stable.



Java Platform, Standard Edition

Java SE 8u31
This release includes important security fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release.
[Learn more](#) ▶














- Installation Instructions
- Release Notes
- Oracle License
- Java SE Products
- Third Party Licenses
- Certified System Configurations
- Readme Files
 - JDK ReadMe
 - JRE ReadMe

JDK
DOWNLOAD ↓

Server JRE
DOWNLOAD ↓


JRE
DOWNLOAD ↓

L'IDE contenant déjà tout le nécessaire pour le développement et la compilation, nous n'avons besoin que du JRE. Une fois que vous avez cliqué sur Download JRE, vous arrivez sur la page représentée à la figure suivante.


Java SE Runtime Environment 8u31		
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.		
<input type="radio"/> Accept License Agreement <input checked="" type="radio"/> Decline License Agreement		
Product / File Description	File Size	Download
Linux x86	40.49 MB	 jre-8u31-linux-i586.rpm
Linux x86	61.57 MB	 jre-8u31-linux-i586.tar.gz
Linux x64	40.62 MB	 jre-8u31-linux-x64.rpm
Linux x64	59.87 MB	 jre-8u31-linux-x64.tar.gz
Mac OS X x64	56.79 MB	 jre-8u31-macosx-x64.dmg
Mac OS X x64	52.71 MB	 jre-8u31-macosx-x64.tar.gz
Solaris SPARC 64-bit	50.86 MB	 jre-8u31-solaris-sparcv9.tar.gz
Solaris x64	48.61 MB	 jre-8u31-solaris-x64.tar.gz
Windows x86 Online	0.61 MB	 jre-8u31-windows-i586-iftw.exe
Windows x86 Offline	29.02 MB	 jre-8u31-windows-i586.exe
Windows x86	51.7 MB	 jre-8u31-windows-i586.tar.gz
Windows x64	89.1 MB	 jre-8u31-windows-x64.exe
Windows x64	54.72 MB	 jre-8u31-windows-x64.tar.gz

2. Eclipse IDE

Choisissez Eclipse IDE for Java Developers, en choisissant la version d'Eclipse correspondant à votre OS (*Operating System* = système d'exploitation), comme indiqué à la figure suivante.



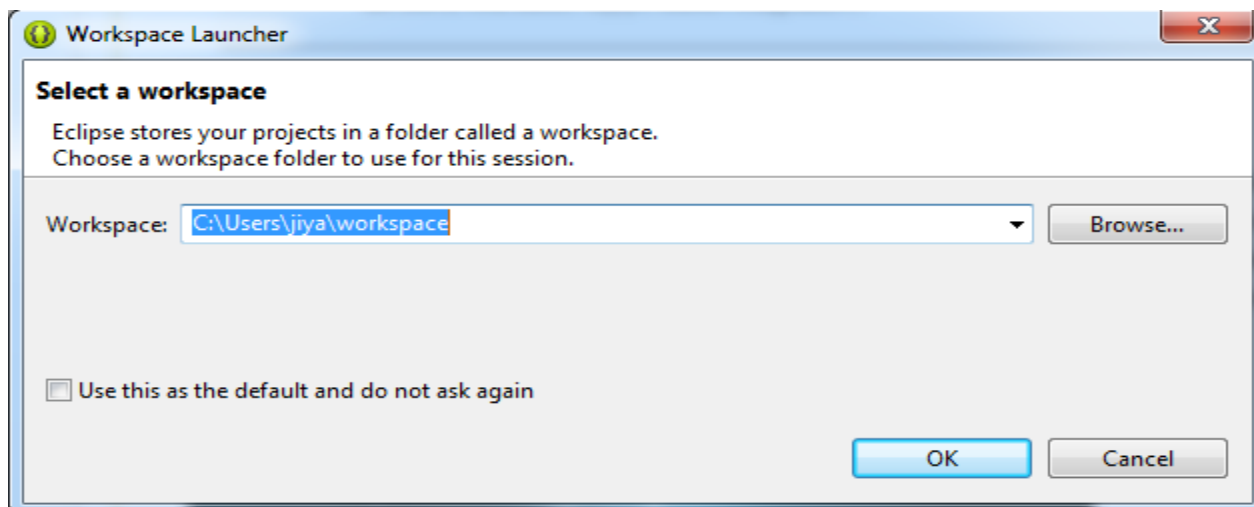
Eclipse IDE for Java Developers, 154 MB
Downloaded 1,409,189 Times
The essential tools for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Mylyn, Maven integration...



Windows 32 Bit
Windows 64 Bit

3. Lancement d'Eclipse IDE

Vous devez maintenant avoir une archive contenant Eclipse. Décompressez-la où vous voulez, entrez dans ce dossier et lancez Eclipse. Au démarrage, comme le montre la figure suivante, Eclipse vous demande dans quel dossier vous souhaitez enregistrer vos projets ; sachez que rien ne vous empêche de spécifier un autre dossier que celui proposé par défaut.



III. Les composants Java Swing

1. Les classes mères Component et JComponent

Les classes graphiques Swing dérivent de la classe JComponent , qui hérite ellemême de la classe AWT (AbstractWindowToolkit).

Tous les composants Swing commencent par la lettre "J" . C'est la principale différence entre les composants AWT et les composants Swing.

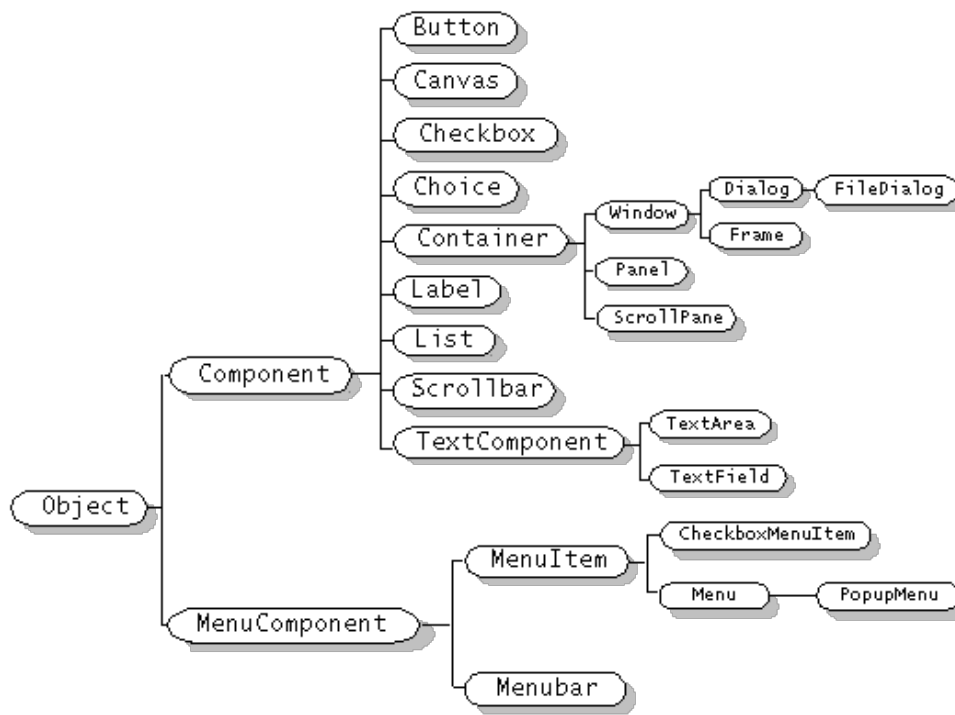
La classe JComponent et les contrôles GUI (Graphical User Interface) se trouvent dans le package javax.swing.

Les composants Swing se répartissent :

- en conteneurs de plus haut niveau (JFrame, JWindow, JApplet et JDialog)
- en conteneurs légers (les autres contrôles GUI Swing).

On peut dire d'une manière générale que :

- Tous les composants de Swing descendent de la classe JComponent.
- La classe JComponent descend de la classe Component de AWT.
- Les composants ont ainsi de tr`es nombreuses m´ethodes communes.



2. La relation entre les composantes

Tous les composants sont capables d'accueillir d'autres composants car ils héritent de la classe Container. On ajoute des composants à l'aide de la méthode add.

3. Les propriétés géométriques

- Les composants peuvent être dimensionnés avec la méthode setSize. Cette méthode est peut utilisée pour imposer une taille aux composants.
- La taille préférentielle d'un composant peut être obtenue grâce à la méthode getPreferredSize.
- La position du composant est définie en utilisant setLocation.
- Le coin supérieur gauche du conteneur parent sert de référence.
- Un composant peut être caché en utilisant la méthode setVisible avec un paramètre booléen.
- Pour désactiver un composant (il devient grisé) on utilise la méthode setEnabled.

Les fenêtres

- Swing propose deux fenêtres différentes : les JWindow et JFrame.
- Elles sont toutes les deux héritières de Window (classe de AWT), elles partagent donc de nombreuses méthodes.
- Pour créer une fenêtre, il suffit simplement d'appeler le constructeur de la classe choisie

```
JFrame fen = new JFrame ();
```

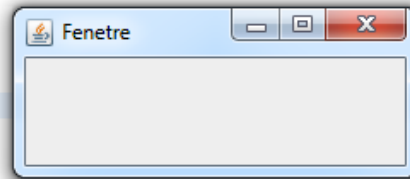
- Lors de la création une fenêtre n'est pas visible, pour la rendre visible, on utilise la méthode setVisible.

Comme pour tous les composants, les méthodes setLocation et setSize sont utilisable pour positionner et dimensionner la fenêtre.

- Pour fixer la taille d'une fenêtre on peut utiliser la méthode pack qui fixe la fenêtre à sa taille idéale (vis-à-vis des composants qui sont à l'intérieur).
- Pour libérer une fenêtre on appelle la méthode dispose, elle est alors libérée par le système d'exploitation puis détruite par le garbage collector.

- Une application graphique est terminée lorsque toutes ses fenêtres sont libérées.
- Les fenêtres JWindow sont des fenêtres simple de l'interface graphique utilisé (Windows, X, ...).
- Elles n'ont ni bordure, ni titre, ni icône.
- Les utilisations principales des JWindow sont les écrans de démarrage ou les écrans d'attente.

```
//Création de la fenêtre
JFrame frame = new JFrame("Fenetre");
//La taille de la fenêtre
frame.setSize(100,100);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//Visibilité
frame.setVisible(true);
```



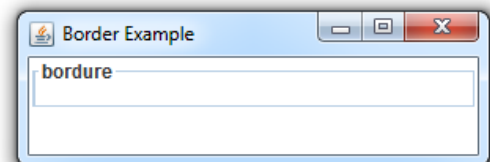
Les conteneurs

- Toute application Swing doit être basée sur au moins un conteneur primaire.
- Il assure le lien entre l'application Java et le système d'exploitation.
- Les conteneurs primaires sont les seuls composants graphiques de Swing construit par le système d'exploitation
- On parle de "composants lourds".

Les bordures

- Les composants peuvent être entourés de bordures.
- Les bordures peuvent être en creux, en relief, avec du texte, ...
- La méthode `setBorder` permet de spécifier une bordure de type `Border`.
- Généralement, le composant `Border` n'est pas instancié grâce aux méthodes statiques de la classe `BorderFactory`.

```
topPanel = new JPanel();
topPanel.setOpaque(true);
topPanel.setBackground(Color.WHITE);
contentPane.setBorder(
    BorderFactory.createEmptyBorder(hgap, hgap, hgap, hgap));
contentPane.setLayout(new BorderLayout(hgap, vgap));
topPanel.setBorder(BorderFactory.createTitledBorder("bordure"));
contentPane.add(topPanel, BorderLayout.PAGE_START);
```



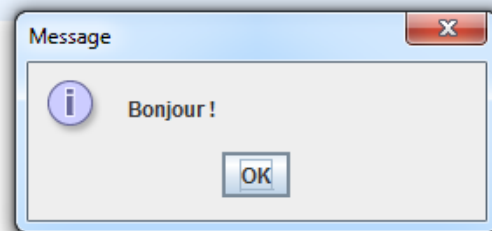
Les boîtes de dialogue

- Les boîtes de dialogue sont utilisées lorsque l'on doit informer l'utilisateur rapidement.
- Une boîte de dialogue est modale (elle bloque l'application en cours).
- Elle comporte un ou plusieurs bouton(s) que l'utilisateur doit utiliser pour valider une action ou un choix.
- La classe `JOptionPane` permet de construire des boîtes de dialogue très facilement.

Il n'est pas nécessaire d'instancier la classe, les principales méthodes étant statiques.

- Pour afficher un message on utilise la méthode `showMessageDialog` de la classe `JOptionPane`.
- Il existe trois versions surchargées de cette méthode, la plus simple utilise deux paramètres : la fenêtre principale de l'application (qui peut être null) et le message à afficher).

```
JOptionPane.showMessageDialog(this,"Bonjour !");
```

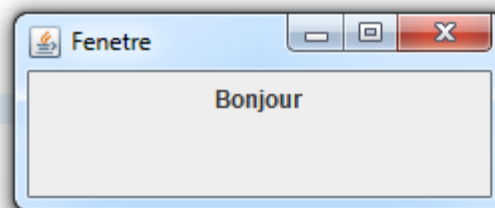


4. Les composants graphiques

Les labels :

Un label est un élément statique d'un interface, on l'utilise pour afficher du texte ou une image. Pour afficher une chaîne de caractères, on peut utiliser la méthode `setText` ou une version surchargée du constructeur.

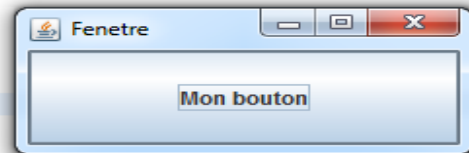
```
//La taille de la fenetre  
setSize(100,100);  
//Création du label  
JLabel l =new JLabel("Bonjour");  
p.add(l);  
this.add(p);  
//Visibilité de fenetre  
setVisible(true);
```



Les boutons :

Les boutons peuvent être configurés, et dans une certaine mesure contrôlés, par des actions. Utilisation d'une action avec un bouton a de nombreux avantages au-delà de la configuration directement sur un bouton.

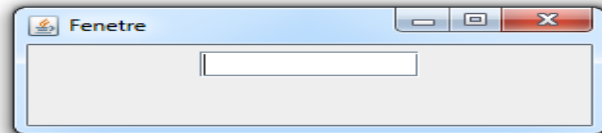
```
//Titre de la fenetre
setTitle("Fenetre");
//La taille de la fenetre
setSize(100,100);
//Création du bouton
JButton b =new JButton("Mon bouton");
this.getContentPane().add(b);
//Visibilité
setVisible(true);
```



Les zones de text :

JTextField est un composant léger qui permet le montage d'une seule ligne de texte. Il est destiné à être une source compatible avec java.awt.TextField où il est raisonnable de le faire. Ce composant a des capacités qui ne les trouve pas dans la classe java.awt.TextField. La superclasse doit être consulté pour des capacités supplémentaires.

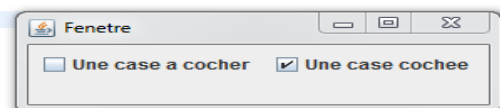
```
JPanel p=new JPanel();
//Titre de la fenetre
setTitle("Fenetre");
//La taille de la fenetre
setSize(100,100);
//Création du zone de text
JTextField t =new JTextField(10);
p.add(t);
this.add(p);
//Visibilité de fenetre
setVisible(true);
```



Les case à cocher :

- Les cases à cocher sont une autre méthode pour représenter un choix booléen.
- Le constructeur permet d'initialiser la chaîne de caractères ou l'état et la chaîne (et un icône...).
- Comme pour les JToggleButton, l'état de la case peut être modifié avec setSelected et lu avec isSelected.

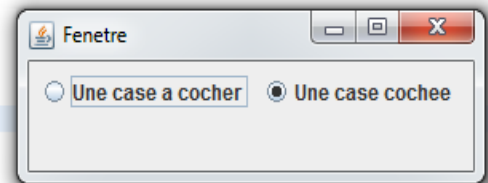
```
setTitle("Fenetre");
//La taille de la fenetre
setSize(100,100);
//Création des cases à cocher
JCheckBox casePasCochee = new JCheckBox ("Une case a cocher ");
JCheckBox caseCochee = new JCheckBox ("Une case cochee ",true );
//L ajout des case acocher au panel
p.add(casePasCochee);
p.add(caseCochee);
this.add(p);
```



Les boutons « radio » :

- Les boutons "radio" sont utilisés pour choisir un élément parmi un ensemble.
- Comme les JToggleButton, ils ont deux états matérialisés et utilisent les mêmes méthodes (setSelected et isSelected).
- Le constructeur permet d'initialiser la chaîne de caractères ainsi que l'état du bouton.

```
//La taille de la fenetre
setSize(100,100);
//Création des cases à cocher
JRadioButton casePasCochee = new JRadioButton ("Une case a cocher ");
JRadioButton caseCochee = new JRadioButton ("Une case cochee ",true );
//L ajout des case acocher au panel
p.add(casePasCochee);
p.add(caseCochee);
this.add(p);
```



- Pour que les boutons "radio" aient un comportement exclusif, ils doivent être groupés dans un BoutonGroup. Ils sont ajoutés en utilisant la méthode add.
- BoutonGroup n'est pas un composant visuel, on matérialise souvent des groupes en plaçant une bordure autour des groupes.
- Les boutons sont placés dans un JPanel autour duquel une bordure est dessinée.

Les listes de choix :

- Les listes de choix permettent de choisir un (ou plusieurs) élément(s) parmi un ensemble.
- Elles utilisent un tableau d'éléments pour représenter les choix possible, ce qui permet une plus grande souplesse que les boutons "radio".
- La méthode setListData permet de définir la liste, il est aussi possible de la passer au constructeur.

```
String [] noms = {" Said ", "Tibari", " nadia ", "jihad " };
//Création de jlist
JList n = new JList ( noms );
p.add(n);
this.add(p);
//Visibilité de fenetre
setVisible(true);
```

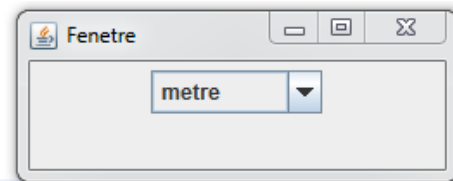


- Le mode de selection est configuré à l'aide de la méthode `setSelectionMode`, plusieurs modes sont possibles
 - ✓ `ListSelectionModel.SINGLE SELECTION` permet de ne sélectionner qu'un seul élément,
 - ✓ `ListSelectionModel.SINGLE INTERVAL SELECTION` permet de sélectionner plusieurs éléments formant un seul intervalle,
 - ✓ `ListSelectionModel.MULTIPLE INTERVAL SELECTION` n'aucune restriction.
- On peut accéder aux indices des éléments avec (`getSelectedIndex` et `getSelectedIndices`) ou aux éléments eux-mêmes (`getSelectedValue` et `getSelectedValues`)).
- Si le nombre d'éléments est grand, on place le composant `JList` dans un `JScrollPane` pour éviter de monopoliser la surface d'affichage.

Les listes «combo»:

- Les boîtes "combo" permettent de choisir un élément parmi un ensemble, l'utilisateur pouvant éventuellement ajouter un élément qui n'est pas proposé.
- Comme les listes de choix, elles utilisent un tableau d'éléments pour représenter les choix possibles.
- Les objets sont ajoutés/supprimés avec les méthodes `addItem` et `removeItem`.
- Il est toutefois possible de passer une liste d'éléments au constructeur.

```
//La taille de la fenetre  
setSize(100,100);  
  
String [] mesure = {" metre " , "kilo" , "temperature" };  
JComboBox m = new JComboBox ( mesure );  
  
p.add(m);  
this.add(p);
```

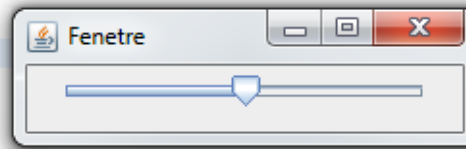


Les glissières :

- Les glissières permettent de matérialiser une valeur d'une manière plus visuelle qu'un chiffre.
- Elles sont souvent utilisées pour représenter des volumes, des niveaux de zoom,...

- Elles peuvent être verticales (JSlider.VERTICAL) ou horizontales (JSlider.HORIZONTAL).

```
//Création d une glissière  
JSlider a = new JSlider ();  
p.add(a);
```

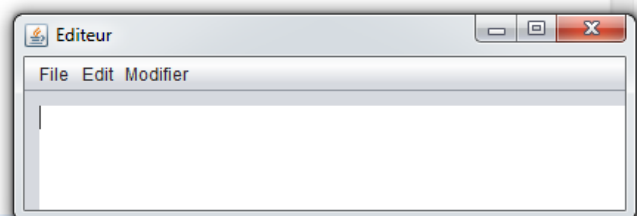


- La position du curseur est accessible par les méthodes getValue et setValue.
- De même, les valeurs des butées sont accessibles par les méthodes getMinimum, setMinimum, getMaximum et setMaximum.
- Les glissières peuvent être agrémentées de graduations visuelles.
- On demande le tracé des graduations avec la méthode setPaintTicks, par défaut les repères ne sont pas tracés.
- Deux types de graduations sont possibles, les majeures et les mineures.
- Leurs valeurs sont accessibles à l'aide de setMajorSpacing et setMinorSpacing.

Le menu :

- Swing propose une approche très hiérarchisée de la construction des menus.
- Les éléments de menus sont des JMenuItem,..., un descendant de AbstractButton.
- Ils sont regroupés dans des menus JMenu à l'aide de la méthode add.
- Les menus sont eux mêmes regroupés dans une barre de menus, JMenuBar
- La barre de menus est soit placée dans un conteneur, soit associée à une JFrame avec setJMenuBar.

```
//la barre  
menuBar = new JMenuBar();  
menuBar.setBounds(100,50,40,20);  
  
//Ajouter le menu au barre de menu  
menu1=new JMenu("File");  
menuBar.add(menu1);  
  
fm2=new JMenuItem("Enregistrer dans un fichier",enregistrer);  
menu1.add(fm2);
```



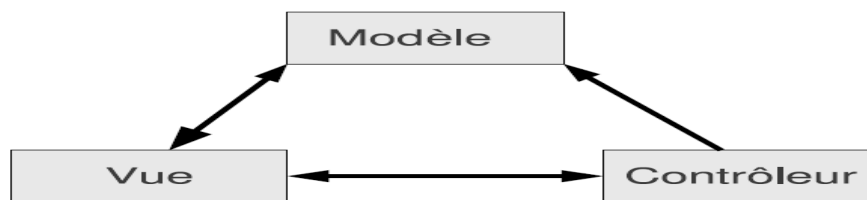
IV. Le Modèle MVC

1. L'architecture MVC

L'architecture Modèle Vue Contrôleur (MVC) est un motif de conception pour le développement d'applications logicielles qui sépare le modèle de données, l'interface utilisateur et la logique de contrôle. Ce motif a été mis au point en 1979 par Trygve Reenskaug, qui travaillait alors sur SmalltalkXerox PARC, dans les laboratoires de recherche

Ce modèle d'architecture impose la séparation entre les données, les traitements et la représentation.

Architecture modèle-vue-contrôleur



Le modèle : Le modèle peut être divers et varié. C'est là que se trouvent les données. Il s'agit en général d'un ou plusieurs objets Java. Ces objets s'apparentent généralement à ce qu'on appelle souvent « la couche métier » de l'application et effectuent des traitements absolument transparents pour l'utilisateur. Par exemple, on peut citer des objets dont le rôle est de gérer une ou plusieurs tables d'une base de données. En trois mots, il s'agit du cœur du programme !

Le contrôleur : Son rôle est de traiter les événements en provenance de l'interface utilisateur et les transmet au modèle pour le faire évoluer ou à la vue pour modifier son aspect visuel (pas de modification des données affichées mais des modifications de présentation (couleur de fond, affichage ou non de la légende d'un graphique, ...)).

Le contrôleur 'connaît' la (les) vues qu'il contrôle ainsi que le modèle. Il pourra appeler des méthodes du modèle pour réagir à des événements (demande d'ajout d'un client par exemple), il pourra faire modifier à la vue son aspect visuel. Il pourra aussi instancier de nouvelles vues (demande d'affichage de telle ou telle info). Pour faire cela, le contrôleur sera à l'écoute d'événements survenant sur les vues. Actuellement, le noyau Java se compose de 12 types d'événements définis dans `java.awt.events`: `ActionEvent`, `AdjustmentEvent`, `ComponentEvent`, `ContainerEvent`

FocusEvent, InputEvent, ItemEvent, KeyEvent, MouseEvent, paintEvent, TextEvent
Et WindowEvent

La vue : Ce que l'on nomme « la vue » est en fait une IHM, elle contient la description de l'interface graphique avec ses différents composants. Elle représente ce que l'utilisateur a sous les yeux.

2. Avantages et inconvénients

Avantages :

Un avantage apporté par ce modèle est la clarté de l'architecture qu'il impose. Cela simplifie la tâche du développeur qui tenterait d'effectuer une maintenance ou une amélioration sur le projet. En effet, la modification des traitements ne change en rien la vue.

Il permettra à plusieurs corps de métier de travailler sur la même application. Par exemple, le Web-designer pourra travailler sur la Vue, le développeur pourra travailler sur le Controller et le DBA pourra travailler sur le Model. Ainsi, tout les acteurs du projet pourront travailler chacun de leur coté sans entrer en conflit avec un tiers.

Il permet aussi d'obtenir un code plus propre et donc plus facile à maintenir par la suite ! En cas de problème visuel, nous savons que nous devons intervenir dans le code de la Vue. En cas de problème concernant une requête, nous savons qu'il faille absolument intervenir dans le code du model. Il change de charte graphique beaucoup plus facilement car seul la Vue sera retouchée.

Inconvénients :

Le MVC se révèle trop complexe pour de petites applications. Le temps accordé à l'architecture peut ne pas être rentable pour le projet.

Même si le code est factorisé, le nombre de micro-composants n'en est pas moins augmenté. C'est le prix à payer pour la séparation des 3 couches. Et toutes les personnes qui font de la gestion de configuration comprendront que le nombre important de fichiers représente une charge non négligeable dans un projet.

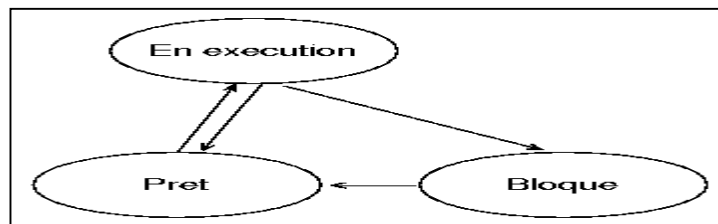
V. Les Threads

1. Introduction

1.1. Rappel sur les processus

Un processus est un programme en cours d'exécution. Pour chaque processus :

- le système alloue de la mémoire:
segment de code (instructions).
 - segment de données (allocation dynamique).
 - segment de pile (variables locales et adresses de retour des fonctions).
-
- Le système sauvegarde un ensemble de registres du processeur que dispose chaque processus lorsque ce dernier n'est pas en cours d'exécution. Un processus peut être dans différents états. On présente d'une manière simplifiée les états possibles :
 - En exécution (*running*) : le processus s'exécute sur un processeur du système;
 - Prêt : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en exécution);
 - Bloqué : il manque une ressource (en plus du processeur) au processus pour qu'il puisse s'exécuter.



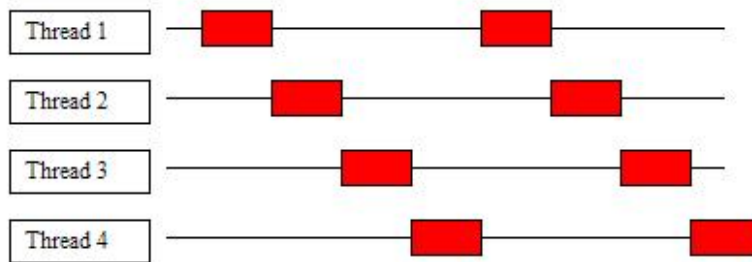
L'exécution de chaque processus nécessite les ressources suivantes :

- Mémoire.
- Processeur.
- Et aussi entrées/sorties (fichiers, communications avec d'autres processus s'exécutant ou non sur la même machine).

1.2. Les Threads

Chaque processus est composé de plusieurs threads. En Java, un thread appelé aussi processus léger qui peut se traduire par «file d'exécution» signifie un point d'exécution dans le programme.

Les threads sont un ensemble d'instructions capable de s'exécuter en parallèle et en des temps partagés à d'autres traitements, et pas au même temps. En outre, les threads d'un même processus partagent le même espace d'adressage, le même environnement (par exemple les mêmes variables d'environnement, des mêmes données, etc.) ce qui rend la communication entre les threads du même processus plus facile que celle entre les processus. Chaque thread possède son propre environnement d'exécution (valeurs des registres du processeur) ainsi qu'une pile (variables locales). Le langage Java est multithread c'est pour cela il est important pour les threads d'interrompre leurs exécutions et de laisser la main aux autres threads pour qu'ils s'exécutent.



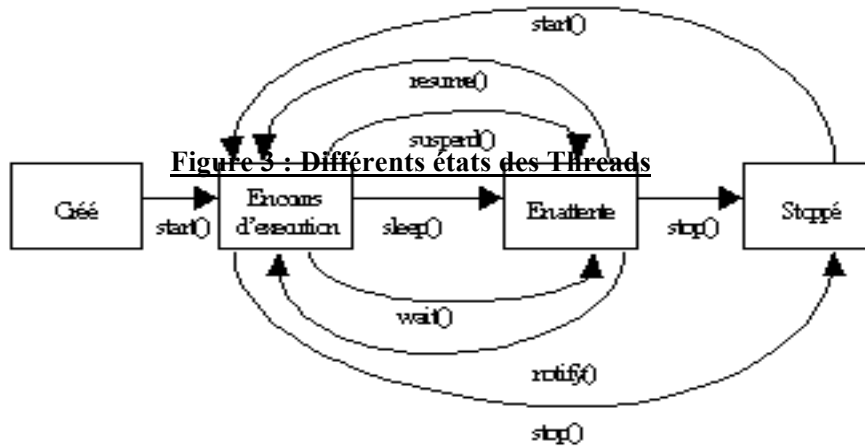
Les rôles et les avantages des threads peuvent être résumés dans les points suivants:

- Le lancement et l'exécution sont plus rapides par rapport aux processus système.
- Simple à utiliser.
- Ressources systèmes partagés.
- Accélère le traitement car il exécute plusieurs instances de code simultanément.
- Dans les sockets les threads peuvent traiter les requêtes de plusieurs clients au même temps.

Les threads font une partie intégrante du langage JAVA. Elles sont créées et gérées à l'aide de la classe **java.lang.Thread**.

2. Cycle de vie d'un Thread

Un thread peut passer par des différents états et qui sont :



Les méthodes de la classe Thread :

Afin de contrôler le comportement des threads, La class Thread offre un certain nombre de méthodes. Dans ce cours on mentionne que les méthodes fréquemment utilisées, quant aux autres méthodes on consultera la documentation du JDK pour plus d'informations.

On présente quelques méthodes dans le tableau suivant :

Méthode	Rôle
void destroy()	met fin brutalement au thread : à n'utiliser qu'en dernier recours.
int getPriority()	renvoie la priorité du thread
ThreadGroup getThreadGroup()	renvoie un objet qui encapsule le groupe auquel appartient le thread
boolean isAlive()	renvoie un booléen qui indique si le thread est actif ou non
boolean isInterrupted()	renvoie un booléen qui indique si le thread a été interrompu
void join()	attend la fin de l'exécution du thread
void join(long)	attend, au plus le nombre de millisecondes fourni en paramètre, la fin de l'exécution du thread
void resume()	reprend l'exécution du thread() préalablement suspendu par suspend(). Cette méthode est dépréciée

action :	Instantiation :	Lancement :
void run()	méthode déclarée par l'interface Runnable : elle doit contenir le code qui sera exécuté par le thread	t1.start();
void sleep(long)	mettre le thread en attente durant le temps exprimé en millisecondes fourni en paramètre. Cette méthode peut lever une exception de type InterruptedException si le thread est réactivé avant la fin du temps.	
void start()	démarrer le thread et exécuter la méthode run()	
void stop()	arrêter le thread. Cette méthode est dépréciée	
void suspend()	suspend le thread jusqu'au moment où il sera relancé par la méthode resume(). Cette méthode est dépréciée	
void yield()	indique à l'interpréteur que le thread peut être suspendu pour permettre à d'autres threads de s'exécuter.	

Tableau 1: quelques méthodes de la classe Threads

3. Création d'un Thread :

Il existe deux façons pour créer un thread :

➤ Méthode 1 : Par héritage de la classe Thread

Cette méthode se fait par la création d'une classe qui dérive de la classe java.lang.Thread. Il suffit alors de redéfinir la méthode run() qui va contenir le traitement à exécuter par le thread.



Exemple :

```

Fenetre.java  Principale.java  Code.java  Bibliotheque.java  Ec
public class MonThread extends Thread {

    public static void main(String[] args) {

        Thread t = new MonThread();
        t.start();
    }

    public void run() {
        int i;

        for (i = 0; i < 10; i++) {
            System.out.println(" " + i);
        }
    }
}

```

CréationInstanciation :Lancement :

➤ **Méthode 2 : Par implémentation de l'interface Runnable**

```
class MonRunnable implements Runnable {
    t1 = new Thread( new MonRunnable());
    t1.start();
}
```

Dans ce cas il suffit d'implémenter l'interface `java.lang.Runnable` et de redéfinir la méthode `run()`. Ensuite il suffit de créer un objet `java.lang.Thread` en lui passant un objet de la classe en paramètre.

Figure 5 : Création de Thread par implémentation de l'interface Runnable

Remarque :

L'instruction de lancement de Threads pour cette méthode lance la méthode `run()` de l'objet référencé par `MonRunnable`.

4. Sommeil d'un Thread

La méthode **`Thread.sleep(long millis)`** bloque le thread qui l'appelle pendant un certain nombre de millisecondes donc d'autres threads à l'état peuvent alors s'exécuter.

Cette méthode met le thread courant en sommeil et elle est appelée dans la méthode `run()` du thread.

Remarque :

- ⇒ Des processus bloquants (entrée/sortie en particulier) peuvent suspendre l'exécution d'un Thread.
- ⇒ Si elle est exécutée dans du code synchronisé, le thread ne perd pas le moniteur (au contra

Comme indiqué dans le tableau qui présente les méthodes de la Classe Thread, on peut vérifier l'état d'un thread en utilisant les méthodes suivantes :

- ❖ **`getState()`** → renvoie l'état du thread.
- ❖ **`isAlive()`** → est vrai si le thread est actif ou endormi.

Exemple :

```

public class MonThread extends Thread{

    public MonThread(String name){super(name);}
    public void run(){
        try{
            System.out.println(this.getName()+" a ete lancé");
            Thread.sleep(100);
            System.out.println(this.getName()+" est terminé");
        }

        catch (InterruptedException e){
            System.out.println(this.getName()+" a été interrompu");
        }
    }

    public static void main(String arg[]){
        try{
            MonThread mt = new MonThread("Test_Sleep");
            mt.start();
            Thread.sleep(100);
            mt.interrupt();
        }
        catch (InterruptedException ie){}
    }
}

```

✓ Résultat :

```

<terminated> MonThread [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (16 févr. 2015 21:19
Test_Sleep a ete lancé
Test_Sleep est terminé

```

5. Synchronisation

Lors de l'exécution des threads en parallèle sur des données qui peuvent être communes, il est donc nécessaire de gérer des conflits d'accès et des problèmes de synchronisation entre les threads.

La synchronisation peut consister à entremêler les exécutions des threads de manière à ce qu'ils ne s'exécutent que chacun à leur tour alternativement. La méthode **join()** permet d'appliquer cette synchronisation.

Exemple :

```

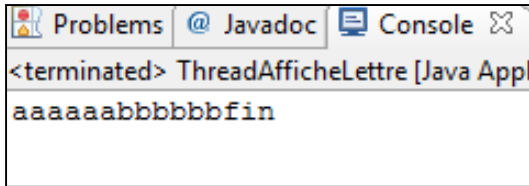
class ThreadAffiche extends Thread{
    private char lettre;
    private int time;
    private int cpt=0;

    public ThreadAffiche(char lettre,int time)
    { this.lettre=lettre; this.time=time;}

    public void run() {
        while(cpt<=time)
        {cpt++;
        System.out.print(lettre); }
    }

    public static void main (String [] args)throws Exception{
        Thread ta=new ThreadAffiche('a',5);
        Thread tb=new ThreadAffiche('b',5);
        ta.start();
        ta.join();
        tb.start();
        tb.join();
        System.out.println("fin");
    }
}

```

✓ **Résultat :**

```
<terminated> ThreadAfficheLettre [Java App]
aaaaaabbabbbfin
```



6. Coordination entre les Threads

✓ **Moniteur d'un objet**

Les moniteurs des objets servent à la protection du code synchronisé contre les accès multiples. En effet, à un moment donné, un seul thread possède le moniteur d'un objet.

Dans le cas où existent d'autres threads qui veulent avoir le même moniteur, ils sont mis en attente, en attendant que le premier thread rende le moniteur.

La coordination entre threads est gérée au sein d'un objet à l'aide des méthodes suivantes :

- **wait()** : qui met le thread en attente infinie jusqu'à ce qu'il soit notifié.
- **notify()** : un thread notifie un seul autre thread afin de mettre fin à son attente.
- **notifyAll()** : la notification s'adresse à tous les threads en attente.

Remarque :

- ✓ Ces méthodes doivent être utilisées dans des méthodes synchronized.

VI. Les Annotations & La Réflexion

1-Introduction

Java SE 5 a représenté les annotations qui sont des métadonnées incluse dans le code source. Elles permettent de mieux documenter le code source, ou plutôt d'utiliser des informations pendant l'exécution d'un programme et interagir avec le compilateur selon ces informations.

Les annotations prennent une place de plus en plus importante dans la plate-forme Java et de nombreuses API open source et leurs utilisations concernent plusieurs fonctionnalités.

Qu'est-ce qu'une métadonnée ?

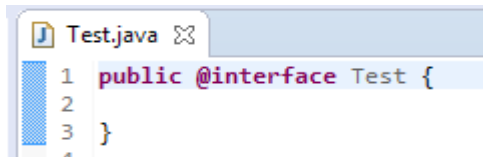
Les métadonnées sont des données s'appliquant à d'autres données en les définissant ou décrivant. Il s'agit de données qui concernent le code, prenons l'exemple des commentaires en JAVADOC qui décrivent le code mais ne changent pas sa signification.

Les métadonnées étaient déjà historiquement utilisées avec Java notamment avec Javadoc ou exploitées par des outils tiers notamment XDoclet .

2- Qu'est-ce qu'une Annotation ?

Une annotation associe des informations arbitraires, ou métadonnées, à un élément de programme Java. Chaque annotation a un nom et de zéro à plusieurs membres, chaque membre a un nom et sa valeur, et ces paires nom=valeur représentent les informations de l'annotation.

La déclaration d'annotation ressemble à la déclaration des interfaces :

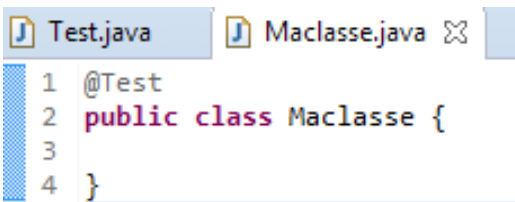


```
Test.java
1 public @interface Test {
2
3 }
4
```

On peut utiliser une annotation sur plusieurs type d'éléments (package, class, interface, énumération, constructeur, paramètres de méthodes, annotations, champs d'une classe ou variable locale).

Il y a la possibilité d'utiliser plusieurs types d'annotation sur le même élément mais on ne peut pas utiliser la même annotation deux fois sur le même élément.

Exemple de mise en œuvre d'une annotation (ici Test) :



```
Test.java  Maclasse.java
1 @Test
2 public class Maclasse {
3
4 }
```

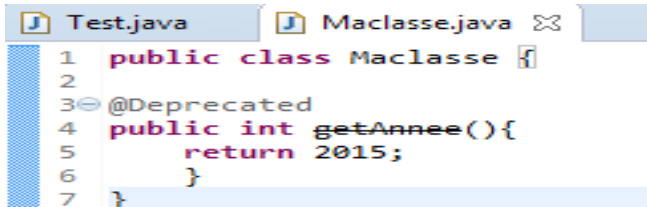
Les Annotations standards :

Les annotations standards sont proposées par Java 5, elles permettent l'interaction avec le compilateur.

☞ L'annotation `@Deprecated` :

L'annotation `@Deprecated` peut être attachée à tout élément. Elle signifie que l'élément annoté est obsolète, et qu'il ne faudrait plus l'utiliser, le compilateur va avertir que l'utilisation de l'élément annoté est découragée.

Exemple d'utilisation de `@Deprecated` :



```

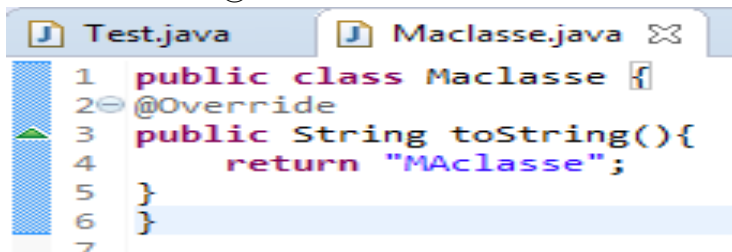
1 public class Maclasse {
2
3 @Deprecated
4 public int getAnnee(){
5     return 2015;
6 }
7

```

☞ L'annotation `@Override` :

L'annotation `@Override` ne peut être utilisée que pour annoter les méthodes afin d'informer le compilateur que cette méthode est redéfinie et qu'il faut cacher une méthode héritée d'une classe parent.

Exemple d'utilisation de `@Override` :



```

1 public class Maclasse {
2 @Override
3 public String toString(){
4     return "MAclasse";
5 }
6 }
7

```

☞ L'annotation `@SuppressWarnings` :

Cette annotation est utilisée pour informer le compilateur d'ignorer les avertissements de la compilation concernant les classes, les méthodes ou les variables. Elle attend un attribut pour marquer le nom d'avertissement qu'il faut supprimer.

Exemple d'utilisation de `@SuppressWarnings` :

```

public class Maclasse {
    @SuppressWarnings("unchecked cast")
    public String toString(){
        return "MAclasse";
    }
}

```

Les Méta-Annotations :

Les méta-annotations sont des annotations utilisées pour annoter d'autres annotations pour indiquer au compilateur comment il doit les traiter.

☞ L'annotation `@Documented` :

La méta-annotation `@Documented` indique au JAVADOC que l'annotation doit être présente dans la documentation générée.

Exemple :

```
public @interface SimpleAnnotation {  
}  
  
import java.lang.annotation.Documented;  
  
@Documented  
public @interface DocAnnotation {  
}
```

La classe ci-dessous Contient deux méthodes annotées par les deux annotations précédentes.

```
public class Exemple {  
  
    /**  
     * Commentaire de la méthode 1.  
     */  
    @SimpleAnnotation  
    public void method1 () {  
        /* ... */  
    }  
  
    /**  
     * Commentaire de la méthode 2.  
     */  
    @DocAnnotation  
    public void method2 () {  
        /* ... */  
    }  
}
```

Lorsqu'on consulte la documentation JAVADOC on ne voit que la deuxième annotation :

Documentation :

```
public void method1()  
  
        Commentaire de la méthode 1.  
  
-----  
  
@DocAnnotation  
public void method2()  
  
        Commentaire de la méthode 2.
```

⚡L'annotation `@Inherited`:

La méta-annotation `@inherited` est utilisée pour annoter les annotations des classes : lorsqu'une classe possède une annotation hérité automatiquement les sous-classes vont recevoir la même annotation.

Exemple :

```
import java.lang.annotation.Inherited;  
@Inherited  
public @interface Test {  
}
```

Toutes les classes qui vont hériter la classe « Maclasse », reçoivent automatiquement l'annotation « Test ».

☞ L'annotation **@Retention** :

La méta-annotation `@retention` indique la « durée de vie » de l'annotation, c'est-à-dire de quelle manière elle doit être générée par le compilateur.

Elle prend trois valeurs :

Valeur	Description
RetentionPolicy.SOURCE	Les annotations ne sont pas enregistrées dans le fichier *.class. Elles ne sont donc accessibles que par des outils utilisant les fichiers sources (compilateur, Javadoc, etc...).
RetentionPolicy.CLASS	Les annotations sont enregistrées dans le fichier *.class à la compilation mais elles ne sont pas utilisées par la machine virtuelle à l'exécution de l'application. Elles peuvent toutefois être utilisées par certains outils qui lisent directement les *.class. Il s'agit du comportement par défaut si la méta-annotation n'est pas présente.
RetentionPolicy.RUNTIME	Les annotations sont enregistrées dans le fichier *.class à la compilation et elles sont utilisées par la machine virtuelle à l'exécution de l'application. Elles peuvent donc être lues grâce à l'API de réflexion

Exemple :

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
}
```

☞ L'annotation **@Target** :

La méta-annotation `@target` est utilisée pour limiter le type d'éléments sur lesquels l'annotation peut être utilisée.

Elle prend ces valeurs :

valeur	Description
ElementType.ANNOTATION_TYPE	L'annotation peut être utilisée sur d'autres annotations.
ElementType.CONSTRUCTOR	L'annotation peut être utilisée sur des constructeurs.
ElementType.FIELD	L'annotation peut être utilisée sur des champs d'une classe.
ElementType.LOCAL_VARIABLE	L'annotation peut être utilisée sur des variables locales.
ElementType.METHOD	L'annotation peut être utilisée sur des méthodes.

ElementType.PACKAGE	L'annotation peut être utilisée sur des packages
ElementType.PARAMETER	L'annotation peut être utilisée sur des paramètres d'une méthode ou d'un constructeur.
ElementType.TYPE	L'annotation peut être utilisée sur la déclaration d'un type : class, interface (annotation comprise) ou d'une énumération (mot-clef enum).

Exemple :

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface Test {
}
```

Cette annotation sera utilisée uniquement pour les constructeurs et les méthodes.

3- Création d'une Annotation

Une simple Annotation @Test :

Nous allons commencer par créer une simple annotation Test :

```
public @interface Test {
```

```
}
```

La deuxième étape est utilisé les méta-annotations standard pour modifier le comportement de notre annotation

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
}
```

Utilisation des attributs :

Pour le moment, on n'a pas utilisé les attributs d'une annotation, les annotations qui n'ont pas des attributs appelées « annotation marqueur ».

```
1 public @interface Test {
2     String value();
3 }
4
```

La méthode « Value() » représente l'attribut de notre annotation.

L'utilisation de notre annotation avec l'attribut « value() » :

```
@Test(value="Notre Annotation ")  
public class Maclasse{  
  
}
```

Ou bien :

```
@Test("Notre Annotation ")  
public class Maclasse{  
  
}
```

Cette dernière déclaration est valable uniquement pour l'attribut « value () »,
Si on a utilisé d'autre attribut comme « attribut » on ne peut pas faire la deuxième déclaration.
Les attributs d'une annotation n'acceptent que les types suivants :

- Un type primitif (boolean, int, float, etc.).
- Une chaîne de caractères (java.lang.String).
- Une référence de classe (java.lang.Class).
- Une Annotation (java.lang.annotation.Annotation).
- Un type énuméré (enum).
- Un tableau à une dimension d'un des types ci-dessus.

Les membres et les valeurs par défaut :

On peut donner à un attribut une valeur par défaut, comme le montre l'exemple ci-dessous :

```
public @interface Test {  
    String num() default "Chaine de caractère";  
}
```

Il est possible qu'une annotation possède plusieurs annotations de mêmes types, il suffit que cette annotation possède un tableau des annotations comme l'exemple ci-dessous :

```
public @interface Tests {  
    Test[] value();  
}
```

Exemple d'utilisation :

```
@Tests({  
    @Test(num="Test1"),  
    @Test(num="Test2")  
})  
public class Maclasse{  
  
}
```

4 -Introduction sur la réflexion

La réflexion (reflection en anglais) permet l'introspection des classes, c'est-à-dire de charger une classe, d'en créer une instance et d'accéder aux membres statiques ou non (appel de méthodes, lire et écrire les attributs) sans connaître la classe par avance.

Java possède une API permettant la réflexion. Elle sert notamment à gérer des extensions (plug-in en anglais) pour une application.

Qu'est-ce qu'une réflexion ?

La réflexion c'est le fait de donner la possibilité au programme de pouvoir accéder et/ou modifier la structure d'un objet pendant l'exécution du programme. Par structure d'un objet, on entend tout ce qui est propriété, classe, fonction etc.

Java implémente la réflexion. Il donne cependant uniquement la possibilité d'accéder aux structures et non pas de les modifier. Ainsi, à l'exécution d'un programme Java, il est tout à fait possible de connaître précisément la structure de n'importe quel objet et de pouvoir interagir avec (c'est à dire par exemple exécuter n'importe quelle fonction de l'objet) mais on ne peut pas par exemple, ajouter une fonction ou un champ à l'objet.

5. La mise en œuvre de la réflexion

Java.lang est un package standards qui regroupe les classes et interfaces fondamentales et possède trois classes utilisées pour l'utilisation dynamique des classes :

java.lang.Class

qui est la classe centrale sur laquelle repose la réflexivité et permet d'accéder à ses caractéristiques .

java.lang.Package

est une classe qui permet d'accéder aux différentes informations d'un package (informations de version, annotations, ...).

java.lang.ClassLoader

est une classe qui permet de gérer le chargement de classe. Il existe des sous-classes dont notamment java.net.URLClassLoader permettant de charger des classes en les cherchant dans une liste d'URLs (donc de fichiers JAR et répertoires également, en convertissant le chemin du fichier ou répertoire en URL).

Les autres classes utiles sont définies dans le package java.lang.reflect et donnent la possibilité d'accéder aux détails d'une classe. Les principales classes sont les suivantes :

java.lang.reflect.Constructor

Référence à un constructeur d'une classe.

java.lang.reflect.Method

Référence à une méthode d'une classe.

java.lang.reflect.Field

Référence à un champ d'une classe.

java.lang.reflect.Modifier

Les attributs et les méthodes statiques pour décoder les modificateurs des membres (public, private, protected, static, abstract, final, native, ...).

Les classes représentant des membres d'une classe (Constructor, Method, Field) implémentent toutes l'interface java.lang.reflect.Member comportant les méthodes suivantes :

Class getDeclaringClass()

Retourne la classe définissant ce membre.

String getName()

Retourne le nom du champ

boolean isSynthetic()

Teste si ce membre a été généré par le compilateur.

int getModifiers()

Retourne un entier codant la visibilité du champ c'est-à-dire public, protected ou private mais également d'autres informations comme static, final, ...

✓ **Chargement dynamique d'une classe :**

la classe Class permet de charger dynamiquement une classe dont le nom pleinement qualifié est fourni en paramètre de l'une de ses deux méthodes statiques :

static Class.forName(String name)

équivalent à appeler la seconde méthode avec pour paramètres : (name, true, this.getClass().getClassLoader()).

static Class.forName(String name, boolean initialize, ClassLoader loader)

charge la classe dont le nom complet est spécifié en utilisant l'instance du chargeur de classe fourni. Le paramètre initialisé vaut true pour initialiser la classe (appeler le bloc d'initialisation statique), ou false pour ne pas l'initialiser.

Il est également possible d'obtenir une java.lang.Class de manière statique :

à partir d'un objet en appelant la méthode **getClass()**, à partir d'une référence à la classe en utilisant le champ class.

Exemple :

```
package ref;
public class Maclasse {
    String nom;
    public String getNom()
    { return nom; }
}

Notre classe « Maclasse »

package ref;

public class Main {

    Class c = Class.forName("ref.Maclasse"); // sans référence statique à la classe
    // ou
    Class c1 = ref.Maclasse.class; // référence statique à la classe
    // ou
    Class c2 = new Maclasse().getClass(); // référence statique à la classe
}
```


✓ **Liste des membres d'une classe :**

Les méthodes suivantes permettent de présenter les membres d'une classe :

getConstructors()

Cette méthode retourne un tableau de `java.lang.reflect.Constructor` qui contient tous les constructeurs définis par la classe.

getMethods()

Cette méthode retourne un tableau de `java.lang.reflect.Method` qui contient tous les constructeurs définis par la classe.

getFields()

Cette méthode retourne un tableau de `java.lang.reflect.Field` qui contient tous les constructeurs définis par la classe.

Les méthodes ci-dessus retournent les membres publics de la classe, comprenant également ceux hérités des classes mères. Il existe une variante "Declared" de ces méthodes retournant tous les membres (publics, protégés, privés) déclarés par la classe uniquement (les membres hérités sont exclus).

Au lieu de lister tous les membres puis en rechercher un en particulier, il est possible d'utiliser les méthodes spécifiques de recherche d'un membre précis d'une classe (publics et hérités, ou bien "Declared" pour tous ceux déclarés par la classe seule) :

getConstructor(Class... parameterTypes)

getDeclaredConstructor(Class... parameterTypes)

Cette méthode retourne une instance de la classe constructor déclaré avec les paramètres dont les types sont spécifiés.

getMethod(String name, Class... parameterTypes)

getDeclaredMethod(String name, Class... parameterTypes)

Cette méthode attend en paramètre un tableau d'objets de type `Class` qui doit contenir les types de chaque paramètre dans l'ordre de leur définition dans la signature du constructeur souhaité et retourne la méthode portant de nom spécifié..

getField(String name)

getDeclaredField(String name)

Cette méthode retourne l'attribut portant de nom spécifié.

Les membres retournés par toutes ces méthodes peuvent être d'instance ou statiques.

La méthode **getAnnotations()** retourne un tableau de `java.lang.Annotation` contenant toutes les annotations associées à la classe.

✓ Instanciation d'une classe et appel d'un constructeur :

La méthode `newInstance()` de la classe `java.lang.Class` permet de créer une nouvelle instance de la classe et d'invoquer son constructeur par défaut :

```
package ref;

public class Main {

    Class c = Class.forName("ref.Maclass");

    Object o = c.newInstance(); // équivaut à new ref.Maclass();
}
```

Une classe comme celle ci-dessous peut ne pas avoir de constructeur sans paramètres :

```
package ref;
public class Maclass {
    String att1;
    int att2;

    public Maclass(String att1, int att2)
    {
        this.att1 = att1;
        this.att2 = att2;
    }
}
```

Dans ce cas, il faut d'abord obtenir le constructeur, puis l'appeler :

```
Class c = Class.forName("ref.Maclass"); // Accès à la classe Maclass
Constructor constr = c.getConstructor(String.class, int.class); // Obtenir le constructeur (String, int)
Object o = constr.newInstance("Programmation Java", 120); // -> new Maclass("Programmation Java", 120);
```

✓ Appel d'une méthode :

Le principe de l'appel d'un constructeur vu juste avant est le même adopté pour l'appel d'une méthode de la classe. Pourtant, pour obtenir la référence à une méthode, il faut spécifier son nom. Lors de l'invocation de la méthode, il faut spécifier l'instance (l'objet) auquel s'applique la méthode (null pour une méthode statique).

Exemple :

```
package ref;
public class Maclass {
    String att1;
    int att2;

    public Maclass(String att1, int att2)
    {
        this.att1 = att1;
        this.att2 = att2;
    }

    public int getNombre(int a){
        return a*10;
    }
}
```

```

Class c = Class.forName("ref.MaClass"); // Accès à la classe MaClass
Constructor constr = c.getConstructor(String.class, int.class); // Obtenir le constructeur (String, int)
Object o = constr.newInstance("Programmation Java", 120); // -> new MaClass("Programmation Java", 120);

Method method = c.getMethod("getNombre", int.class); // Obtenir la méthode getNombre(int)
int nb = (int)method.invoke(o, 2); // -> o.getNombre(2);

```

✓ Accès à un attribut :

L'accès à un attribut se fait en appelant les méthodes sur l'instance de `java.lang.reflect.Field` obtenu auprès de la classe.

Exemple :

```

Class c = Class.forName("ref.MaClass"); // Accès à la classe MaClass

Constructor constr = c.getConstructor(String.class, int.class); // Obtenir le constructeur (String, int)
Object o = constr.newInstance("Programmation Java", 120); // -> new Livre("Programmation Java", 120);

Field att1 = c.getField("att1"); // Obtenir l'attribut titre
String attribut= (String)att1.get(o); // -> o.att1
att1.set(o, "Java"); // -> o.att1= "Java";

```

✓ L'interface Type:

Avant de définir cette fameuse interface `Type` Il ne faut pas qu'il nous échappe de signaler que par défaut, les méthodes de réflexion retournent les types représenté par le compilateur et non les types paramétrés spécifiés par le programmeur. Alors Les méthodes de réflexion ont deux versions : celle utilisant la classe `Class` pour représenter les types par exemple ex: `getExceptionTypes()` et la deuxième paramétré qui utilise l'interface `Type` prenons à titre d'exemple: `getGenericsExceptionTypes()`.

L'interface `Type` est une interface de base de tout les types Java :

✓ Les types paramétrés : ParameterizedType

Les types paramétrés servent à faire abstraction des types de données. Les collections d'objets sont les structures de données les plus adaptées à leur utilisation et à leur compréhension.

```

List<Double> lis = new ArrayList<Double>();
lis.add(new Double(2.5d));
Double d = lis.get(0);

```

Ses méthodes :

- `Type[] getActualTypeArguments()` : Renvoi types d'arguments (ici, `Double`)
- `Type getRawType()` : Renvoi type Raw (ici, `List`)
- `Type getOwnerType()` : Renvoi le type englobant (pour les classes internes) ou null.

➤ Les wildcard, WildcardType

`WildcardType` : le wildcard est avec une borne qui est soit supérieur soit inférieur

- `Type[] getLowerBounds()` : renvoie les bornes inférieures

- Type[] getUpperBounds() : renvoie les bornes supérieures
 - GenericArrayType est un tableau de type paramétré ou de variable de type
 - Type getGenericComponentType() : renvoie Le type contenu du tableau
 - Les classes : Class :
 - TypeVariable<T> :
- Avec T correspondant au type sur lequel il est déclaré
- String getName() : revoie le nom de la variable.
 - Type[] getBounds() : revoie les bornes de la variable.
 - T getGenericDeclaration() : renvoie l'élément déclarant (Class, Method ou Constructor)
 - Les tableaux de type paramétré
- (List<String>[]) ou devariable de type (T[]), GenericArrayType.

✓ **Les tableaux par réflexion:**

Java.lang.reflect.Array propose Java.lang.reflect.Array qui permet la création et/ou la manipulation des tableaux de type primitif ou des tableaux d'objet à l'aide des méthodes suivante

- **static Object newInstance(Class componentType, int length)** : Permettant la création d'un tableau.
- **static Object get*(Object array, int index)** : Permettant l'obtention de la valeur d'une case spécifiée par son index « index ».
- **static void set*(Object array, int index, Object value)** : Permettant le changement de la valeur d'une case

Voici un exemple : De création d'un tableau multidimension et d'application des getters et setters pour obtention et pour modification de ses cases.

```

1 import java.lang.reflect.Array;
2 import static java.lang.System.out;
3
4 public class CreationTab{
5     public static void main(String[] args){
6         //Création d'un tableau multi dimension par réflexion
7         Object matrice = Array.newInstance(int.class, 2, 2);
8         //Obtention en utilisant get
9         Object i0 = Array.get(matrice, 0);
10        Object i1 = Array.get(matrice, 1);
11        //modification en utilisant set
12        Array.setInt(i0, 0, 1);
13        Array.setInt(i0, 1, 2);
14        Array.setInt(i1, 0, 3);
15        Array.setInt(i1, 1, 4);
16
17        for (int i = 0; i < 2; i++)
18            for (int j = 0; j < 2; j++)
19                out.format("Matrice[%d][%d] = %d\n", i, j, ((int[][])matrice)[i][j]);
20    }
21 }

```

Console

```

<terminated> CreationTab [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (12 févr. 2015 13:41:16)
Matrice[0][0] = 1
Matrice[0][1] = 2
Matrice[1][0] = 3
Matrice[1][1] = 4

```

6. Réflexion et Sécurité:

La réflexion ne tient pas compte des modificateurs d'accessibilité (private, protected, public ou package). Il est donc possible d'accéder à des éléments déclarés privés. Pour remédier à ce problème Java propose des gestionnaires de sécurité qui fournissent des droits et possèdent des permissions, l'une d'entre elles étant d'autoriser la réflexion. Alors Par défaut, la réflexion vérifie la sécurité lors de l'exécution, on ne peut effectuer les opérations que si l'on a les droits. Exemple :

```
1 import java.lang.reflect.Constructor;
2 import java.lang.reflect.InvocationTargetException;
3
4
5 public class MonMain {
6
7
8     public static void main(String[] args) throws NoSuchMethodException,
9         InvocationTargetException, InstantiationException,
10        IllegalAccessException {
11        // essai d'accès au constructeur privé
12        Constructor<String> c=String.class.getDeclaredConstructor(int.class,int.class,char[].class);
13        char[] array="Bonjour".toCharArray();
14        String name=c.newInstance(0,array.length,array);
15    }
16 }
17
18
```

Exception in thread "main" java.lang.IllegalAccessException: Class MonMain can not access a member of class java.lang.String with modifier

at sun.reflect.Reflection.ensureMemberAccess(Unknown Source)
at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(Unknown Source)
at java.lang.reflect.AccessibleObject.checkAccess(Unknown Source)
at java.lang.reflect.Constructor.newInstance(Unknown Source)
at MonMain.main(MonMain.java:16)

Mais si l'on veut passer outre la sécurité et accélérer notre code on peut utiliser la méthode `setAccessible` qui permet d'éviter de faire le test de sécurité. `setAccessible` est une méthode de `AccessibleObject` dont héritent les `Constructor`, `Field` et `Method`. Voici un exemple de son utilisation :

```
1 import java.lang.reflect.Constructor;
2 import java.lang.reflect.InvocationTargetException;
3
4
5 public class MonMain {
6
7
8     public static void main(String[] args) throws NoSuchMethodException,
9         InvocationTargetException, InstantiationException,
10        IllegalAccessException {
11        // accès possible avec SetAccessible()
12        Constructor<String> cons=String.class.getDeclaredConstructor(
13        int.class,int.class,char[].class);
14        cons.setAccessible(true);
15        char[] Tab="Annotation & Réflexion".toCharArray();
16        String text=cons.newInstance(0,Tab.length,Tab);
17        System.out.println(text); // Annotation & Réflexion
18        Tab[4]='\n';
19        System.out.println(text); // Annotation & Réflexion
20    }
21 }
22
23
```

Console

<terminated> MonMain [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (10 févr. 2015 14:13:43)
Annotation & Réflexion
Annotation & Réflexion

VII. La spécification JPA

1. ORM (Object Relational Mapping):

Avant d'entamer le JPA et ses différentes sections, il faut en premier lieu comprendre la notion d'ORM.

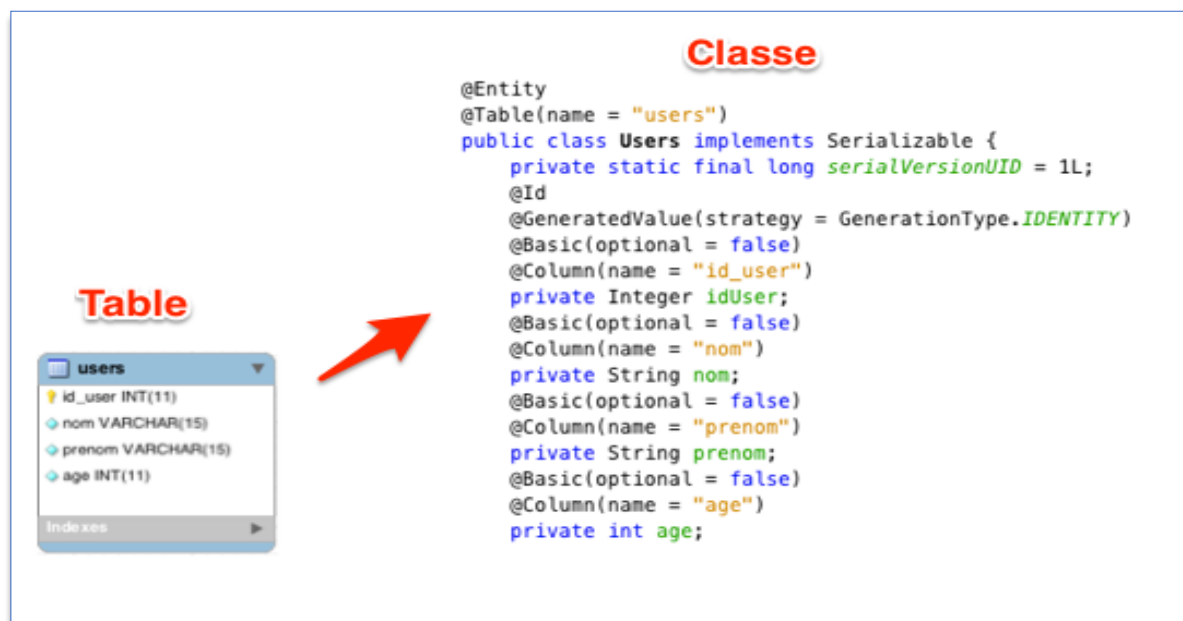
Puisque plusieurs applications nécessitent une couche de persistance pour stocker ses différentes données dans un SGBD, les organiser et les récupérer par la suite. Une technique de programmation faisant la liaison entre les bases de données relationnelles et la programmation objet aidera à faire la gestion des données persistantes dans le SGBD et va résoudre la différence entre le modèle Objet et le modèle Relationnel.

Cette technique est l'ORM, elle 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.

Même si 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 avec divers frameworks tel que Hibernate, TopLink...

Voici un exemple d'une classe (Users) avec ses attributs qui correspond à une table (Users) de la base de donnée avec ses champs.



QU'EST CE QUE LE JPA ?

JPA (Java Persistence API) est un API standard offert par JavaEE/JavaSE et considéré comme un Framework ORM simplifiant le modèle de persistance. L'utilisation de cet API ne requiert aucune ligne de code mettant en oeuvre l'API JDBC.

L'API Java Persistence repose sur des entités qui sont de simples POJOs (Plain Old Java Objects) annotées et sur un gestionnaire de ces entités (EntityManager) qui propose des fonctionnalités pour les manipuler (ajout, modification suppression, recherche). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

JPA est basé aussi sur un jeu d'interfaces et de classes permettant de séparer l'utilisateur d'un service de persistance (l'application) et le fournisseur d'un service de persistance, un jeu d'annotations standardisées pour préciser la mise en correspondance entre classes Java et tables relationnelles, un fournisseur de persistance (par exemple Hibernate), un fichier XML « persistence.xml » décrivant les moyens de la persistance (fournisseur, datasource, etc.)

Cette technologie est utilisable dans les applications Web (conteneur Web), ou dans les EJB (serveur d'applications) ou bien dans les applications standard (Java Standard Edition).

- Nous traiterons dans la suite du cours les notions suivantes : Les entités, les associations, les comportements dans JPA, les requêtes et nous terminerons avec une mise en oeuvre pour appliquer cet ensemble de notions et mieux comprendre le fonctionnement du JPA.

2. Les Entités

Introduction : Afin que JPA puisse faire le lien entre un objet Java et la base de données, les classes à mapper sont définies en tant qu'entité JPA. Une classe d'entité JPA est un POJO (Plain Old Java Object) classe, c'est à dire une classe Java ordinaire qui est marqué (annoté) comme ayant la capacité de représenter des objets dans la base de données. Conceptuellement, cela est similaire aux classes sérialisables, qui sont marquées comme ayant la capacité d'être sérialisé.

Une entité JPA est, par définition, une classe Java qui doit avoir les propriétés suivantes :

- Elle doit posséder un constructeur vide, public ou protected. Ce constructeur vide existe par défaut si aucun constructeur n'existe dans la classe. Dans le cas contraire, il doit être ajouté explicitement.
- Elle ne doit pas être final, et aucune de ses méthodes ne peut être final.
- Une entité JPA ne peut pas être une interface ou une énumération.
- Une entité JPA peut être une classe concrète ou abstraite.

Une classe possède des champs. Ces champs sont en général associés à des valeurs en base, rangées dans les colonnes d'une table. L'objet EntityManager est capable de positionner lui-même les valeurs de ces champs. Il le fait par injection, soit directement sur le champ, soit en passant par les getters / setters. Ce point n'est pas spécifié de façon très satisfaisante en JPA 1.0, et a été revu en JPA 2.0.

○ Annotation de l'entité

Les principales annotations que l'on peut mettre sur la déclaration d'une classe sont `@Entity`, `@Table` et `@Access`.

-L'annotation `@Entity` nous indique que cette classe est une classe persistante. Elle peut prendre un attribut `name`, qui fixe le nom de cette entité. Par défaut, le nom d'une entité est le nom complet de sa classe.

-L'annotation `@Table` permet de fixer le nom de la table dans laquelle les instances de cette classe vont être écrites. Cette annotation est particulièrement utile lorsque l'on doit associer un jeu de classes à des tables existantes. L'annotation `@Table` supporte plusieurs attributs :

Les attributs `catalog`, `schema` et `name` : permettent de fixer les paramètres de la table utilisée.

L'attribut `@UniqueConstraints` permet d'écrire des contraintes d'unicité sur des colonnes ou des groupes de colonnes.

-L'annotation `@Access` permet de fixer la façon dont l'entity manager va lire et écrire les valeurs des champs de cette classe. Cette annotation ne prend qu'un seul attribut, qui ne peut prendre que deux valeurs : `AccessType.FIELD` et `AccessType.PROPERTY`. Dans le premier cas, les lectures / modifications se font directement sur les champs, dans le second elles passent par les getters / setters. Notons que cette annotation peut aussi être placée sur les champs ou getters d'une classe. On peut donc particulariser l'injection de valeur champ par champ.

Exemple :

```
@Entity
@Table(name = "produit",
        uniqueConstraints={@UniqueConstraint(
            name="nom_Prod", columnNames={"nom_produit"}
        )})
@XmlRootElement
public class Produit implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    -
```



Notons que l'on peut mettre autant de contraintes `@UniqueConstraint` que l'on veut, séparées par des virgules.

Dans notre exemple, le nom de l'entité n'est pas fixé, on peut tout de même le fixer en ajoutant « `@Entity(name=produit)` ». La table dans laquelle les instances de cette classe seront écrites se nomme « produit ». Nous avons écrit une contrainte d'unicité, nommée `nom_prod`, qui impose de ne pas avoir deux produits qui portent même nom.

○ Annotations des champs

D'une façon générale, les champs d'une classe sont automatiquement associés à des colonnes en base, dans la table de l'entité. De plus, ces colonnes portent les mêmes noms que ces champs. Si cette règle par défaut suffit, alors il n'y a pas à annoter les champs.

Les types associés automatiquement à des colonnes sont les suivants :

Tous les types de base Java : int, long, float, etc...

Tous les types enveloppes (wrappers) des types de base, et le type String.

Les types date java.sql.Date, java.sql.Time et java.sql.Timestamp. Les types java.util.Date et java.util.Calendar ne sont donc pas associés automatiquement.

Les types énumérés.

Les types Serializable, qui sont rangés dans des blob.

Il arrive très fréquemment que l'on doive particulariser le type d'application entre un champ d'une classe et une colonne d'une table. Il est notamment très fréquent que le nom choisi par défaut ne convienne pas, on doit donc le fixer à la main. On dispose pour cela de deux annotations : @Column et @Lob. Le cas de l'annotation @Basic, qui s'applique aux champs sera traité dans la suite.

a- Annotation @Column

Cette annotation peut se poser sur un champ ou sur un getter. Cette position détermine la façon dont l'entity manager injecte les valeurs dans les instances de cette classe.

L'annotation @Column expose les attributs optionnels suivants :

name : permet de fixer le nom de la colonne.

length : pour les chaînes de caractères, cet attribut permet de fixer la taille du champ SQL associé. Notons que cette taille n'est pas vérifiée dans le code Java.

unique : ajoute une contrainte d'unicité sur la colonne.

nullable : empêche cette colonne de porter des valeurs nulles. Cette contrainte est utile pour la définition de clés étrangères.

insertable, updatable : empêche la modification des valeurs de cette colonne, utilisée pour les clés étrangères.

precision, scale : fixe les paramètres d'échelle et de précision des colonnes qui portent des nombres.

columnDefinition : permet de donner, en SQL, le code de création d'une colonne. Cet attribut est à utiliser avec précautions, dans la mesure où il impose d'écrire du code SQL natif dans le code Java. Ce code SQL est propre à la base de données cible. C'est précisément ce que l'on ne veut pas faire lorsque l'on fait du JPA !

b- Annotation @Lob

Cette annotation permet de forcer l'application de cette colonne dans un blob SQL.

c- Cas des colonnes temporelles

Comme nous l'avons vu, les champs de type java.util.Date et java.util.Calendar ne sont pas associés avec des données en base directement. Pour préciser le type SQL temporel avec lequel on veut stocker ces champs, on dispose d'une annotation spécifique : @Temporal. Cette annotation prend un unique attribut,

qui peut prendre les valeurs `TemporalType.DATE`, `TemporalType.TIME` ou `TemporalType.TIMESTAMP`.

- **Cas des colonnes énumérées**

Les énumérations Java peuvent être associées à des colonnes de deux façons. Soit l'entity manager crée une colonne numérique, et stocke dedans un entier, qui correspond au numéro d'ordre de la valeur que porte le champ, soit l'entity manager crée une colonne de type chaîne de caractères, et stocke le nom de la valeur énumérée.

On spécifie ce point avec l'annotation `@Enumerated`, qui peut prendre deux valeurs en attribut : `EnumType.ORDINAL` ou `EnumType.STRING`.

3. Les associations (mapping)

Introduction : Les associations sont une des parties les plus importantes dans l'utilisation de JPA. C'est elles en effet qui permettent de définir les relations entre la base de données relationnelle et les entités qui sont des POJOs¹. Le mécanisme de mapping² est très simplifié par l'utilisation d'annotations. Cette section d'écrit comment décrire réaliser le mapping en utilisant les annotations JPA2.

a. Bidirectionnel ou unidirectionnel

Les relations entre entités, telles que définies en JPA peuvent être unidirectionnelles ou bidirectionnelles. La différence fondamentale entre bidirectionnel et unidirectionnel est que dans une relation bidirectionnelle, les deux entités contiennent des attributs de référence annotés entre eux, alors que dans le cas d'une relation unidirectionnelle, une seule des entités possède une référence sur l'autre. Dans les relations unidirectionnelles l'une des deux entités doit être maître et l'autre esclave. Propriétaire d'une association

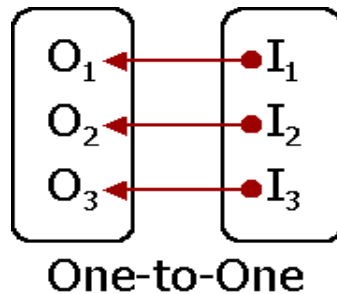
Toute relation a un côté propriétaire. C'est ce qui détermine des mises à jour de la relation dans une base de données. Dans le cas d'une relation bidirectionnelle, l'un des EntityBeans est le propriétaire de la relation et l'autre est appelé le côté inverse. Dans une relation unidirectionnelle, du fait qu'un seul des Entity Beans dans la relation possède une référence sur l'autre entité, il est de fait possesseur de la relation (Owner side) et il n'y a pas de côté inverse. Concrètement pour les relations OneToOne et ManyToMany, on peut choisir le côté maître comme on le souhaite. Dans le cas des relations OneToMany et ManyToOne, l'entité du côté « One » est l'entité esclave.

¹ POJO POJO est un acronyme qui signifie Plain Old Java Object que l'on peut traduire en français par bon vieil objet Java. Il est principalement utilisé pour faire référence à la simplicité d'utilisation d'un objet Java en comparaison avec la lourdeur d'utilisation d'un composant EJB.

² Le terme association est rarement utilisé dans le monde Java. Il est plus courant qu'il soit désigné par « mapping ».

b. One-to-one

Cette annotation est utilisée lorsque deux tables doivent avoir entre elle une cardinalité de 1 ver 1 dans la base de données relationnelle. Les entités peuvent être associées avec une relation one-to-one en utilisant l'annotation `@OneToOne`. Il y a plusieurs manières de réaliser les associations one-to-one:



- Les entités associées partagent les mêmes valeurs de clef primaire, une clé étrangère est détenue par l'une des entités (notez que cette colonne FK dans la base de données devrait être contrainte d'unicité pour simuler un à une multiplicité).
- Une table d'association est utilisé pour stocker le lien entre les deux entités (une contrainte unique doit être définie sur chaque FK pour assurer la cardinalité un à un).

Tout d'abord, nous traçons une véritable association one-to-one en utilisant des clefs primaires partagées:

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }
    @OneToOne(cascade = CascadeType.ALL)
    @PrimaryKeyJoinColumn
    ...
}
```

L'annotation `@PrimaryKeyJoinColumn` indique que la clé primaire de l'entité est utilisée comme valeur de clé étrangère de l'entité associée.

```
@Entity
public class Heart {
    @Id
    public Long getId() { ... }
}
```

Dans l'exemple suivant, les entités associées sont liées à travers une colonne de clé étrangère explicite.

```
@Entity
public class Client implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="passeport_fk")
    public Passeport getPassport() {
        ...
    }
}

@Entity
public class Passeport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
        ...
    }
}
```

Un client est lié à un Passeport, avec une colonne de clef étrangère nommée `passport_fk` dans la table client. La colonne de jointure est déclarée avec l'annotation `@JoinColumn` semblable à l'annotation `@Column`. Il a cependant un paramètre de plus nommé `@referencedColumnName`. Ce paramètre déclare la colonne dans l'entité cible qui sera utilisé pour la jointure.



Notez que lors de l'utilisation `@referencedColumnName` à une colonne autre que la clef primaire, la classe associée doit être `Serializable`. A noter également que le `@referencedColumnName` à une colonne autre que la clef primaire doit être mappé sur une propriété ayant une seule colonne.

L'association également peut être bidirectionnelle. Dans une relation bidirectionnelle, l'un des côtés (et seulement un) doit être le propriétaire: le propriétaire est responsable de la colonne de l'association (s) de mise à jour. Pour déclarer une extrémité comme non responsable de la relation, l'annotation `@mappedBy` est utilisé. `@mappedBy` fait référence au nom de la propriété de l'association sur le côté du propriétaire. Dans notre cas, c'est `passport`. Il n'y a pas de déclaration de colonne de jointure puisqu'elle a déjà été déclarée du côté du propriétaire.

```
@Entity
public class Client implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name = "ClientPasseports",
        joinColumns = @JoinColumn(name="client_fk"),

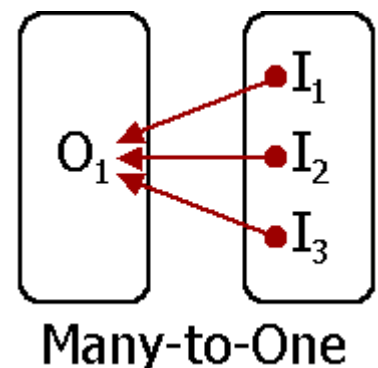
        inverseJoinColumns = @JoinColumn(name="passport_fk"))
    )
    public Passport getPassport() {
        ...
    }
}

@Entity
public class Passeport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Client getOwner() {
        ...
    }
}
```

Un client est lié à un Passeport à travers une association table nommée `ClientPasseports`. Cette table d'association a une colonne de clé étrangère nommée `passport_fk` pointant vers la table `Passeport` matérialisée par `@inverseJoinColumn`, et une colonne de clef étrangère nommée `customer_fk` pointant vers la table `Customer` matérialisée par `@joinColumns`.

c. Many-to-one

Il s'agit là d'association d'un unique élément à une autre classe d'entité qui plusieurs éléments. Il n'est normalement pas nécessaire de préciser l'entité cible explicitement car il peut généralement être déduit du type de l'objet référencé. Si la relation est bidirectionnelle, le côté non propriétaire `OneToMany` entité doit utiliser l'élément `mappedBy` pour spécifier le champ de la relation ou de la propriété de l'entité qui est propriétaire de la relation.



Les associations Many-to-one sont déclarées au niveau propriété avec l'annotation `@ManyToOne`:

```
@Entity()
public class Vol implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinColumn(name="COMP_ID")
    ...
}
```

L'attribut `@JoinColumn` est facultatif. `@ManyToOne` a un paramètre nommé `targetEntity` qui décrit le nom de l'entité cible. Lui aussi est rarement utilisé car la configuration par défaut est en général suffisante.

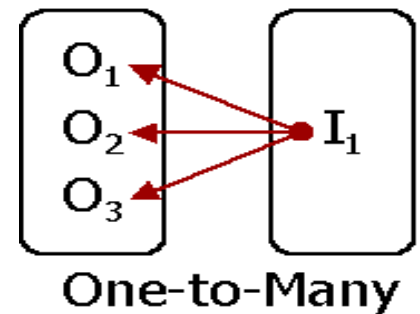
```
@Entity
public class Vol implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE},
targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
    ...
}

public interface Compagnie {
    ...
}
```

Il est également possible de mapper une association ManyToOne avec une table d'association. Cette association décrite par l'annotation `@JoinTable` qui contient une clé étrangère référençant la table de l'entité qui lui est associée.

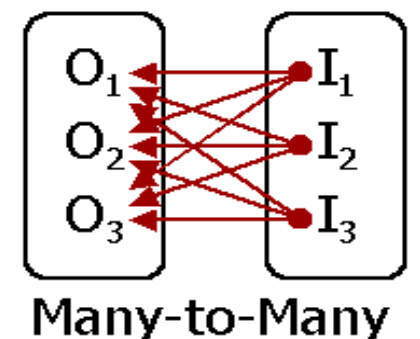
d. One-to-many

Une relation 1:p est caractérisée par un champ de type Collection dans la classe maître. La classe esclave ne porte pas de relation retour. Cette relation peut être spécifiée soit par l'annotation `@OneToMany`. Elle est semblable à l'association ManyToOne. La seule différence réside dans le côté. Car en réalité pour une relation ManyToOne, lorsqu'on se place du côté « One » on obtient une relation OneToMany.



e. ManyToMany

Cette annotation définit une association de valeurs multiples avec plusieurs-à-plusieurs multiplicités. Chaque association ManyToMany a deux côtés : le côté propriétaire et le côté inverse. La table de jointure est spécifiée du côté propriétaire. Si l'association est bidirectionnelle, chaque côté peut être désigné comme le côté propriétaire.



```

@Entity
public class Client {
    @Id
    private int idClient;
    @ManyToMany
    Collection<Telephone> noTels;
    ...}

@Entity
public class Telephone{
    private int idTelephone;
    @ManyToMany(mappedBy="noTels")
    Collection<Client> clients;
    ...}

```

L'exemple ci-dessus illustre un cas de relation ManyToMany. Chaque client peut avoir plusieurs numéros de téléphone et un numéro de téléphone peut appartenir à plusieurs clients.

Il est également possible d'utiliser l'annotation `@JoinTable` si le mapping par défaut ne convient pas.

L'annotation `@JoinTable` permet de préciser les paramètres suivants:

- donne des informations sur la table association qui va représenter l'association
- attribut **name** donne le nom de la table
- attribut **joinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté propriétaire de l'association
- attribut **inverseJoinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté qui n'est pas propriétaire de l'association

f. L'héritage

JPA offre plusieurs manières de mapper la relation d'héritage entre des classes. Il revient donc au développeur de faire son choix en fonction des exigences de son cahier de charges.

✓ Heritage table unique (SINGLE_TABLE)

L'ensemble de la hiérarchie d'héritage est mappé dans une seule même table. C'est-à-dire que la classe mère et les classes fille sont stockées dans une seule table de la base de données relationnelle qui contiendra à la fois les attributs de la classe mère et ceux de la classe fille. Tous les champs de la table ne sont pas utilisés lorsque la classe mère est stockée. Un champ supplémentaire « discriminator value » permet de faire la différence entre les deux classes cela implique donc qu'une colonne supplémentaire soit rajoutée elle est désignée par « discriminator column ».

```

@Entity
@Inheritance
@DiscriminatorColumn(name="PROJET_TYPE")
@Table(name="PROJET")
public abstract class Project {
    @Id
    private long id;
    ...}

@Entity
@DiscriminatorValue("P")
public class PFE extends Project {
    private BigDecimal budget;
}

@Entity

```

```
@DiscriminatorValue("M")
public class MiniProjet extends Project {
}
```

Dans cet exemple la classe mère Projet possède deux classes filles PFE et MiniProjet. La distinction entre les deux types de projet dans la base de données se fera par la colonne « PROJET_TYPE ». pour une ligne qui stocke un projet de type PFE la valeur enregistrée est P. M sera utilisé pour les projets de type « MiniProjet ».



La classe mère est abstraite. Il ne faut donc pas l'instancier. Il faudra toujours passer par les classes filles pour créer un Objet ou le stocker dans la base de données.

✓ Heritage table par classe (TABLE_PER_CLASS)

Chaque classe d'entité concrète dans la hiérarchie est mappée à une table séparée. Un objet est stocké dans exactement une ligne dans la table spécifique à type. Cette table spécifique contient colonne pour tous les champs de la classe, y compris les champs hérités. Cela signifie que les frères et sœurs dans une hiérarchie d'héritage auront chacun leur propre copie des champs qu'ils héritent de leur super-classe.

```
@Entity
@Table(name="Personne")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Personne {
    ...
}

@Entity
@Table(name="Etudiant")
public class Etudiant extends Magazine {
    ...
}
```

Dans cet exemple la classe étudiant hérite de la classe personne et dispose de sa propre table de base de données.

✓ Heritage avec jointure (JOINED)

Chaque classe dans la hiérarchie est représentée comme une table séparée. Cependant il n'y a pas de duplication des champs hérités pour les classes filles. Un objet est stocké étalé sur plusieurs tables. Une ligne dans chacune des tables qui composent sa hiérarchie d'héritage de classe. Cela signifie que les attributs que la classe doit hériter sont stockés dans la table qui correspond à la classe mère et les autres attributs sont stockés dans une table propre à chacune des tables filles.

```
@Entity
@Table(schema="DOCUMENT")
@Inheritance(strategy=InheritanceType.JOINED)
public class Document {
    ...
}

@Entity
@Table(name="LINE_ITEM", schema=" DOCUMENT")
@PrimaryKeyJoinColumn(name="ID", referencedColumnName="ID")
public static class Contrat extends Document { ... }
```

Dans l'exemple ci-dessus les tables document et Contrat sont jointes par l'attribut « ID ». La classe document constitue la classe mère de laquelle hérite la classe Contrat.

4. Le Comportement

a) La persistance par transitivité

Quand un objet est persisté dans la base de données, cela implique implicitement que les objets référencés par cet objet devraient être rendus persistants, ce qui est un comportement logique car un objet n'est pas vraiment persistant si une partie des valeurs de ses propriétés n'est pas persistante et c'est ce qu'on appelle la persistance par transitivité.

Mais ce n'est pas si simple que ça ! Parce que si par exemple on supprime cet objet persistant, comment va t-on maintenir la cohérence automatique des autres objets qu'il a rendu persistants à travers la persistance par transitivité ? Et est ce qu'on doit les supprimer de même ? Si c'est le cas, le service de persistance doit donc examiner toutes les références des objets qui sont rendus persistants, et ce de façon récursive, ce qui peut nuire aux performances.

Par défaut JPA n'effectue pas la persistance par transitivité, parce que rendre un objet persistant ne suffit pas à rendre automatiquement et immédiatement persistants tous les objets qu'il référence, ce qui le rend plus souple.

Et puisque cette cohérence n'est pas gérée automatiquement, elle doit se faire avec du code, c'est là où la possibilité d'avoir la persistance par transitivité dans JPA se fait avec l'attribut « cascade » dans les informations de mapping de l'association.

b) Le comportement CASCADE

Le comportement « cascade » consiste à appliquer les effets d'une opération sur les entités associées à un objet, les valeurs possibles que peut prendre cet attribut sont PERSIST, REMOVE, REFRESH, MERG et ALL qui correspond à toutes ces opérations.

On peut trouver l'attribut « cascade » aussi dans les annotations qui décrivent les associations entre objets, pour indiquer que certaines opérations doivent être appliquées aux objets associés.

Exemple :

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "idCat")  
private Collection<Produit> produitCollection;
```

c) Suppression des entités orphelines

JPA 2.0 a apporté une nouveauté par rapport aux versions précédentes en ce qui concerne la détection et la suppression automatique des orphelins (les entités n'ayant pas d'entité mère), ce qui n'a pas de sens dans la plupart du temps parce que normalement il faut qu'il y ait des associations entre les entités. Ceci se fait à l'aide d'un attribut défini sur @OneToOne et @OneToMany qui est orphanRemoval. Quand on met ce dernier à true cela activera la détection d'entités orphelines, et leur effacement automatique.


```

@Entity
public class Cake implements Serializable {
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nom;
    @OneToMany(mappedBy = "departement")
    private List<Employe> employees;
    @OneToOne(cascade = CascadeType.REMOVE; orphanRemoval=true)
    private Employe employe;
    // ...
}

```

d) Récupération des entités associées

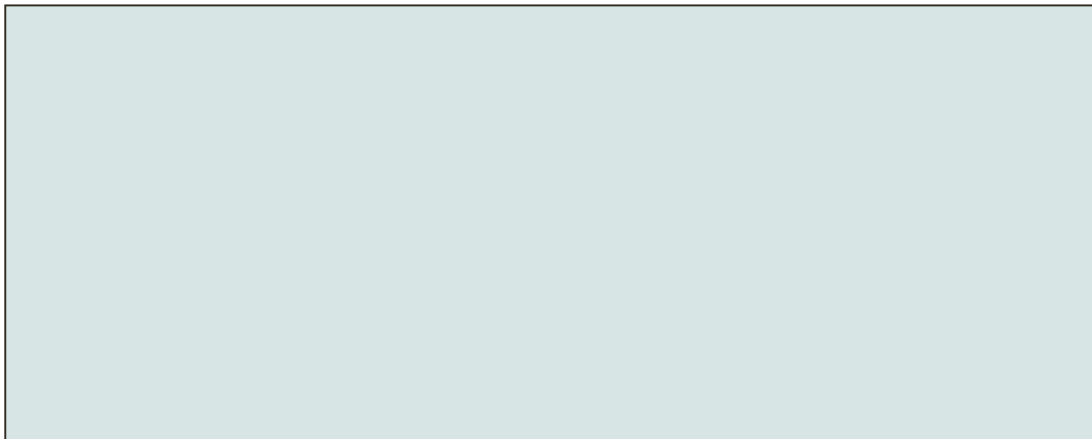
Après avoir traité le problème de la suppression des entités associées, vient un autre qui concerne leur récupération depuis la base de données que ça soit par une requête ou la fonction prédéfinie "find" : si l'on récupère donc une entité, est ce que les entités qui y sont associées vont elles être récupérées aussi? Si c'est oui ceci peut engendrer un autre problème qui est la récupération de plusieurs entités qui ne seront pas utiles pour le traitement en cours. Afin d'éviter ce problème, JPA laisse le choix de récupérer ou non immédiatement les entités associées et ceci en choisissant entre les deux modes de récupération LAZY et EAGER sans influencer la requête. Quelle est donc la différence entre ces deux modes?

LAZY : Dans ce mode, les données associées ne sont récupérées que lorsque c'est vraiment nécessaire. Par contre dans le mode EAGER, la récupération est immédiate.

Pour mieux comprendre la différence entre les deux modes, on vous présente l'exemple suivant :

Exemple Supposons que l'on a une relation entre les tables Département et Employé dans notre modèle de données. Et puisque qu'un département peut avoir plusieurs employés, les attributs que peut avoir la table Département sont : _id, nom, et une collection d'employés qui appartiennent au département.

L'association donc à partir de l'entité Département sera One-To-Many et à partir de l'entité Employé



Maintenant, si on veut récupérer une entité de département de la base de données, JPA peut les récupérer de deux manières différentes :

```
public class Employee {  
    @Id  
    private Long id;  
    private String prenom;  
    private String nom;  
    @ManyToOne()  
    @JoinColumn(name = "employee_id")  
    private Department department;  
    ...  
}
```

Lazy : cela veut dire que JPA va récupérer seulement l'id et le nom, et si l'on veut récupérer la liste des employés, on doit utiliser le getter getEmployees() dans la classe Département.

Si le département comporte plusieurs employés, il sera inutile de les récupérer tous si on n'a pas besoin d'eux. Donc, pour spécifier le mode "Lazy", on doit modifier dans l'annotation de OneToMany comme suit :

@OneToMany(fetch = FetchType.LAZY)

Pour le cas du mode Eager, JPA récupère toutes les propriétés d'un département, y inclus tous les employés associés. Pour le faire, on spécifie dans l'annotation OneToMany annotation comme suit :

@OneToMany(fetch = FetchType.EAGER)



Note : En JPA, par défaut :

- Le mode EAGER est affecté aux associations @OneToOne et @ManyToOne.
- Le mode LAZY est affecté aux associations @OneToMany et @ManyToMany.

e) Les Interfaces principales

Afin d'utiliser le JPA, on se trouve toujours dans la nécessité d'utiliser quelques interfaces qui sont indispensables pour son fonctionnement, ces interfaces sont les suivantes :

❖ EntityManagerFactory:

- C'est une interface qui est utilisée par l'application afin de créer des Entity Managers.
- Une seule entity manager factory par unité de persistance.
- Quand l'application n'utilise plus l'entityManagerFactory, ou quand elle s'arrête, l'application doit fermer l'entity manager factory. Une fois elle est fermée, toutes les entity managers seront considérées fermées aussi.

❖ - EntityManager:

Cette interface assure les interactions entre la base de données et les beans entités, elle permet de lire et rechercher des données mais aussi de les mettre à jour (ajout, modification, suppression). L'EntityManager est donc au coeur de toutes les actions de persistance.

L'EntityManager assure aussi les interactions avec un éventuel gestionnaire de transactions.

❖ Query, TypedQuery

En JPA2, les requêtes sont représentées par deux interfaces :

- Query : L'ancienne interface, qui était la seule disponible pour effectuer des requêtes pour le JPA1.
- TypedQuery : Une nouvelle interface introduite en JPA 2, elle étend de l'interface Query, elle est utilisée surtout quand on s'attend à un type spécifique pour la valeur de retour.

Ces Interfaces sont utilisées pour la configuration des requêtes, la définition des paramètres et l'exécution des requêtes.

Nous traiterons dans le chapitre suivant plus profondément les requêtes.

4. Les Requêtes

Introduction :

Le langage JPQL est un langage de requête dont la grammaire est définie par la spécification J.P.A.

Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. On trouve également le nom d'EJBQL dans la littérature Java, il s'agit du nom donné à ce langage dans la norme EJB2.

Le langage de requête utilise les schémas de persistance abstraite des entités, y compris leurs relations, pour son modèle de données et définit les opérateurs et les expressions sur la base de ce modèle de données. La portée d'une requête couvre les schémas abstraits d'entités connexes qui sont emballés dans la même unité de persistance. Le langage de requête utilise une syntaxe de type SQL pour sélectionner des objets ou des valeurs basées sur les types de schéma abstrait de l'entité et des relations entre eux.

a) Création de requêtes à l'aide du Java Persistence Query Language

Les méthodes EntityManager.createQuery et EntityManager.createNamedQuery sont utilisées pour interroger la banque de données à l'aide de des requêtes JPQL.

La méthode CreateQuery est utilisée pour créer des requêtes dynamiques, qui sont des requêtes définies directement dans le logique métier d'une application:

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")  
        .setParameter("custName", name)  
        .setMaxResults(10)  
        .getResultList();  
}
```

La méthode `createNamedQuery` est utilisée pour créer des requêtes statiques, ou des requêtes qui sont définis dans les métadonnées en utilisant l'annotation `javax.persistence.NamedQuery`. L'élément de nom d'`@NamedQuery` spécifie le nom de la requête qui sera utilisé avec la méthode de `createNamedQuery`. L'élément d'interrogation de `@NamedQuery` est la requête:

```
@NamedQuery(  
    name="findAllCustomersWithName",  
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"  
)
```

Voici un exemple de `createNamedQuery`, qui utilise le `@NamedQuery`:

```
@PersistenceContext  
public EntityManager em;  
...  
customers = em.createNamedQuery("findAllCustomersWithName")  
    .setParameter("custName", "Smith")  
    .getResultList();
```

✓ Paramètres nommés dans les requêtes:

Les paramètres nommés sont des paramètres de la requête qui sont préfixés par deux points (:). Les paramètres nommés dans une requête sont liés à un argument de la méthode suivante:

```
javax.persistence.Query.setParameter(String name, Object value)
```

Dans l'exemple suivant, l'argument de nom à la méthode métier `findWithName` est lié à « : custName » paramètre nommé dans la requête en appelant `Query.setParameter`:

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")  
        .setParameter("custName", name)  
        .getResultList();  
}
```

Les paramètres nommés sont sensibles à la casse et peuvent être utilisés par les deux requêtes dynamiques et statiques.

✓ Paramètres de position dans les requêtes :

Vous pouvez utiliser les paramètres de position au lieu de paramètres nommés dans les requêtes. Les paramètres positionnels sont précédés d'un point d'interrogation (?), Suivi de la position numérique du paramètre dans la requête. La méthode `Query.setParameter` (position de nombre entier, la valeur de l'objet) est utilisé pour définir les valeurs des paramètres.

Dans l'exemple suivant, la méthode métier `findWithName` est réécrite pour utiliser les paramètres d'entrée:

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")  
        .setParameter(1, name)  
        .getResultList();  
}
```

b) La syntaxe simplifiée du langage de requêtes

Cette section décrit brièvement la syntaxe du langage de requête afin que vous puissiez passer rapidement à Exemple de requêtes:

Les requêtes SELECT:

Une requête SELECT a six articles: SELECT, FROM, WHERE, GROUP BY, HAVING et ORDER BY. Les clauses SELECT et FROM sont nécessaires, mais le WHERE, GROUP BY, HAVING et ORDER BY clauses sont facultatives.

Voici la syntaxe de haut niveau d'une requête Select du langage de requête :

```
select_clause from_clause  
[ where_clause ] [ groupby_clause ] [ having_clause ] [ orderby_clause ]
```

La clause SELECT définit les types des objets ou des valeurs renvoyées par la requête.

La clause FROM définit la portée de la requête en déclarant une ou plusieurs variables d'identification, qui peuvent être référencées dans les clauses SELECT et WHERE. Une variable d'identification représente l'un des éléments suivants :

Le nom de schéma abstrait d'une entité

Un élément d'une relation de collection

Un élément d'une relation à valeur unique

Un membre d'une collection qui est le côté multiple d'une relation un -à-plusieurs

La clause WHERE est une expression conditionnelle qui restreint les objets ou valeurs récupérées par la requête. Bien que la clause soit facultative, la plupart des requêtes ont une clause WHERE.

GROUP BY groupes clause résultats de la requête conformément à un ensemble de propriétés.

La clause HAVING est utilisée avec la clause GROUP BY pour restreindre davantage les résultats de la requête selon une expression conditionnelle.

La clause ORDER BY trie les objets ou les valeurs renvoyées par la requête dans un ordre précis.

c) Exemples de requêtes

Les requêtes suivantes sont de l'entité Player.

✓ **Requêtes simples :**

Si vous n'êtes pas familier avec le langage de requête, ces requêtes simples sont un bon endroit pour commencer.

Une requête Select de base :

```
SELECT p  
FROM Player p
```

Les données récupérées : Tous les joueurs.

Description: La clause FROM déclare une variable d'identification du nom de p, en omettant le mot-clé AS. Si le mot-clé AS a été inclus, la clause serait rédigée comme suit:

```
FROM Player AS p
```

L'élément Player est le nom de schéma abstrait de l'entité Player.

L'élimination des valeurs en double :

```
SELECT DISTINCT p  
FROM Player p  
WHERE p.position = ?1
```

Les données récupérées : Les joueurs avec la position spécifiée par le paramètre de la requête.

Description: Le mot-clé DISTINCT élimine les valeurs en double.

La clause WHERE limite les joueurs récupérés en vérifiant leur position, un champ persistant de l'entité du joueur. L'élément ? 1 désigne le paramètre d'entrée de la requête.

✓ **Requêtes Avec les paramètres nommés :**

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

Les données récupérées : Les joueurs ayant les positions et les noms spécifiés.

Description: Les éléments de position et le nom des champs persistants de l'entité du joueur. La clause WHERE compare les valeurs de ces champs avec les paramètres nommés de la requête, définie à l'aide de la méthode Query.setNamedParameter. Le langage de requête représente un paramètre nommé d'entrée en utilisant un deux-points (:) suivi par un identificateur. Le premier paramètre d'entrée est : position, la seconde est : nom.

✓ **Une requête accédant à des entités liées :**

Dans le langage de requête, une expression peut traverser, ou naviguer, pour les entités liées. Ces expressions sont la principale différence entre le langage SQL et Java Persistencequery. Les requêtes accèdent à des entités liées, alors que SQL joignent les tables.

Une requête simple avec les relations :

```
SELECT DISTINCT p
FROM Player p, IN(p.teams) t
```

Les données récupérées : Tous les joueurs qui appartiennent à une équipe

Description: La clause FROM déclare deux variables d'identification : p et t. La variable p représente l'entité du joueur, et la variable t représente l'entité de l'équipe liée. La déclaration de t référence la variable p précédemment déclarée. Le mot-clé IN signifie que les équipes est une collection d'entités apparentées. L'expression de p.teams navigue d'un joueur à son équipe lié.

Vous pouvez également utiliser l'instruction JOIN pour écrire la même requête :

```
SELECT DISTINCT p
FROM Player p JOIN p.teams t
```

Cette requête pourrait également être réécrite comme suit:

```
SELECT DISTINCT p
FROM Player p
WHERE p.team IS NOT EMPTY
```

Naviguer à valeur unique du champ de la relation :

Utilisez l'instruction clause JOIN pour accéder à une valeur unique de champ de la relation :

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR l.sport = 'football'
```

Dans cet exemple, la requête renvoie toutes les équipes qui sont soit des équipes de soccer ou des équipes de football.

Traversant des relations avec un paramètre d'entrée :

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

Les données récupérées : Les joueurs dont les équipes appartiennent à la ville indiquée.

Description: Cette requête est similaire à l'exemple précédent, mais ajoute un paramètre d'entrée. Le mot clé AS dans la clause FROM est optionnel. Dans la clause WHERE, la période qui précède la ville de variable persistante est un séparateur, pas un opérateur de navigation. Strictement parlant, les expressions peuvent accéder à des champs de relation (entités liées), mais pas à des champs persistants. Pour accéder à un champ persistant, une expression utilise la période comme un séparateur.

Les expressions ne peuvent pas naviguer au-delà des champs de relation qui sont des collections. Dans la syntaxe de l'expression, un champ de collecte à valeur est un symbole terminal. Parce que le domaine des équipes est une collection, la clause WHERE ne peut pas préciser p.teams.city.

Traversant des relations multiples :

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

Les données récupérées : Les joueurs qui appartiennent à la ligue spécifié.

Description: Les expressions dans cette requête naviguer sur deux relations. L'expression de p.teams navigue la relation Player -Team, et t.league expression navigue la relation Team -League.

Dans les autres exemples, les paramètres d'entrée sont des objets String, dans cet exemple, le paramètre est un objet dont le type est une ligue. Ce type correspond au champ de leaguerelease dans l'expression de comparaison de la clause WHERE.

Navigation selon les champs partagés:

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

Les données récupérées : Les joueurs qui participent dans le sport spécifié.

Description: Le champ persistant sport appartient à l'entité de la Ligue. Pour atteindre le terrain de sport, la requête doit d'abord naviguer à partir de l'entité du joueur de l'équipe (p.teams), puis de l'équipe à l'entité de la Ligue (de t.league) . Parce que ce n'est pas une collection, le champ de la relation de la ligue peut être suivie par le domaine sport persistant.

✓ **Les requêtes avec d'autres expressions conditionnelles :**

Chaque clause WHERE doit spécifier une expression conditionnelle, dont il existe plusieurs types. Dans les exemples précédents, les expressions conditionnelles sont des expressions de comparaison qui testent pour l'égalité. Les exemples suivants montrent d'autres types d'expressions conditionnelles. Pour obtenir une description de toutes les expressions conditionnelles, voir la clause WHERE.

L'expression «LIKE »:

```
SELECT p
FROM Player p
WHERE p.name LIKE 'Mich%'
```

Les données récupérées : Tous les joueurs dont les noms commencent par " Mich ".

Description: L'expression LIKE utilise des caractères génériques pour rechercher des chaînes qui correspondent au modèle générique . Dans ce cas la requête utilise l'expression LIKE et le caractère générique% à trouver tous les joueurs dont le nom commence par la chaîne " Mich " Par exemple , " Michael " et "Michelle" deux correspondent au modèle générique .

L'expression IS NULL:

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

Les données récupérées : Toutes les équipes ne sont pas associées à une ligue.

Description: L'expression IS NULL peut être utilisée pour vérifier si une relation a été fixée entre deux entités. Dans ce cas, la requête vérifie si les équipes sont associées à toutes les ligues et renvoie les équipes qui n'ont pas une ligue.

L' expression IS EMPTY:

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

Les données récupérées : Tous les joueurs qui n'appartiennent pas à une équipe.

Description: Le champ de l'entité du joueur de la relation des équipes est une collection. Si un joueur ne fait pas partie d' une équipe , la collection équipes est vide , et l'expression conditionnelle est VRAI .

L'expression BETWEEN:

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

Les données récupérées : Les joueurs dont les salaires situer dans la fourchette des salaires déterminés.

Description: BETWEEN expression a trois expressions arithmétiques : un terrain persistante (p.salary) et les deux paramètres d'entrée (: lowerSalaryet : higherSalary) . L'expression suivante est équivalente à l'expression BETWEEN :

```
p.salary > = : lowerSalary AND p.salary < = : higherSalary
```

Mises à jour et suppression :Les exemples suivants montrent comment utiliser l'instruction UPDATE et DELETE expressions dans les requêtes. UPDATE et DELETE fonctionnent sur plusieurs entités selon la ou les conditions fixées dans la clause WHERE. La clause WHERE dans requêtes UPDATE et DELETE suit les mêmes règles que les requêtes SELECT.

Mise à jour:

```
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```


Description: Cette requête définit l'état d'un ensemble de joueurs comme inactifs si le dernier jeu du joueur est plus long que la date indiquée inactiveThresholdDate.

Suppression:

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

4. Exemples de Mise en œuvre

INSTALLATION :

Si vous travaillez avec Eclipse , alors vous devez télécharger EclipseLink et ajouter les jars suivants à votre projet :

- * eclipselink.jar
- * javax.persistence_*.jar

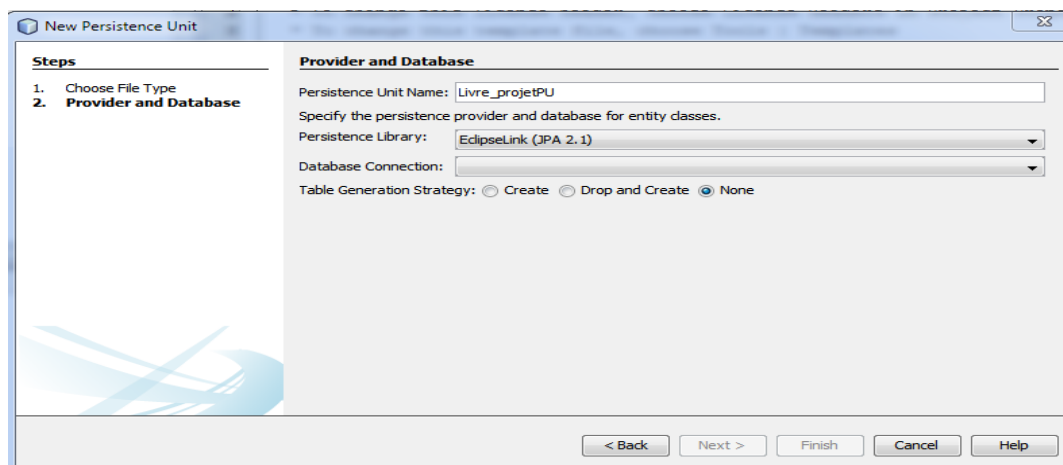
Au cas où vous travaillez avec Netbeans , les jars sont inclus ,vous n'aurez pas besoin de les ajouter

A) PREMIER EXEMPLE

Dans cet exemple nous allons travailler avec une table « Livre » dans notre base de données :

Livre	
<u>id livre</u>	<u>Entier</u>
nom	Caractère (100)
categorie	Caractère (30)
édition	Caractère (50)
auteur	Caractère (30)
Identifiant_1	

Nous commençons par la création d'un projet Java Simple , puis on crée Java Persistent Unit dans notre projet (Netbeans).



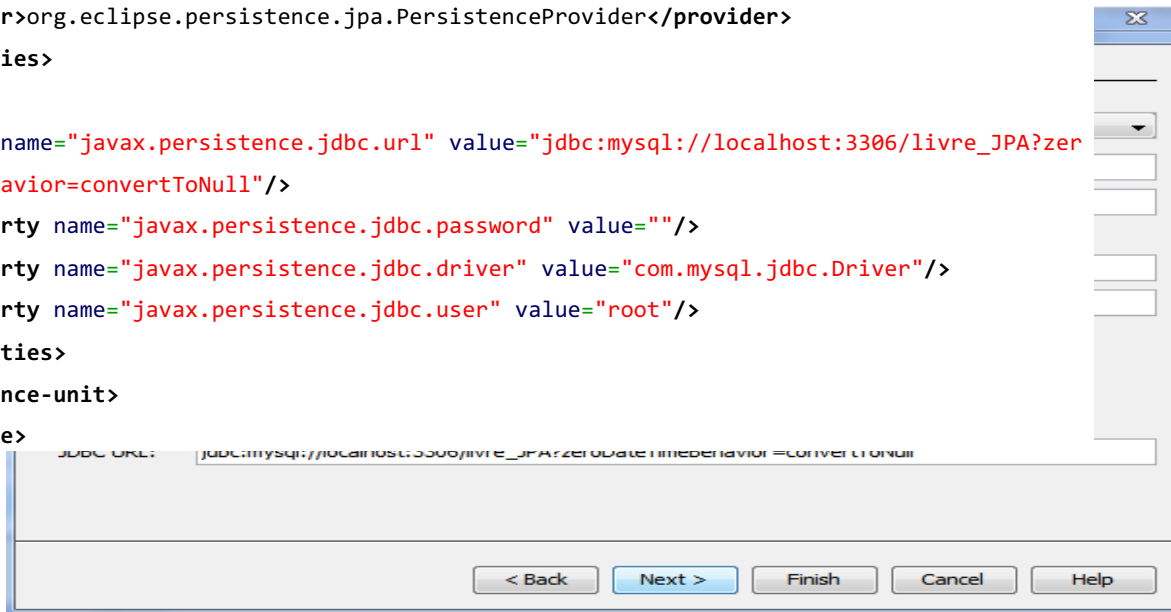
```

1 version="1.0" encoding="UTF-8"?>
instance version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://w
.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
//xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
rsistence-unit name="Livre_projetPU" transaction-type="RESOURCE_LOCAL">
provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
properties>

operty name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/livre_JPA?zer
TimeBehavior=convertToNull"/>
<property name="javax.persistence.jdbc.password" value=""/>
<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
<property name="javax.persistence.jdbc.user" value="root"/>
/properties>
ersistence-unit>
sistence>

```

es appels , Nous



NB : Testez la connexion avec la base de données avant de créer Persistent Unit.

Un fichier « persistence.xml » sera crée après cette opération :

tity

```
ic class Livre implements Serializable {
private static final long serialVersionUID = 1L;
private Integer idLivre;
private String nom;
    private String categorie;
    private String edition;
    private String auteur;
public Livre() {
}
public Livre(Integer idLivre) {
    this.idLivre = idLivre;}

blic Livre(Integer idLivre, String nom, String categorie, String edition, String auth
{
onstructeur ..

public Integer getIdLivre() {return idLivre; }
public void setIdLivre(Integer idLivre) {this.idLivre = idLivre; }
lic String getNom(){return nom; }
public void setNom(String nom){ this.nom = nom; }

blic String getCategorie() { return categorie; }
public void setCategorie(String categorie) { this.categorie = categorie; }
public String getEdition() { return edition; }
public void setEdition(String edition) { this.edition = edition; }
public String getAuteur() { return auteur; }
public void setAuteur(String auteur) {this.auteur = auteur; }
@Override
public String toString() {
    return "livre_projet.Livre[ idLivre=" + idLivre + " ]";
}
```

IES from DATABASE
e classe

```
public class Main {  
    private static final String PERSISTENCE_UNIT_NAME = "Livre_projetPU";  
    private static EntityManagerFactory factory;  
  
    public static void main(String[] args) {  
        factory = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);  
        EntityManager em = factory.createEntityManager();  
        // Recuperer tous Les Livres de la base de données  
        Query q = em.createQuery("select * from Livre ");  
        List<Livre> Livre_list = q.getResultList();  
        for (Livre lv : Livre_list) {  
            System.out.println(lv);  
        }  
        System.out.println("Size: " + Livre_list.size());  
  
        // Creation d'un nouveau Livre  
        em.getTransaction().begin();  
        Livre book = new Livre();  
        book.setNom("test");  
        book.setCategorie("test");  
  
        book.setAuteur("test");  
        book.setEdition("test");  
  
        em.persist(book);  
        em.getTransaction().commit();  
  
        em.close();
```

Entity

```

public class MaisonEdition implements Serializable {
    private static final long serialVersionUID = 1L;
    private Integer idMaison;
    private String nomMaison;

    @OneToMany(mappedBy = "maisonEdition")
    private List<Livre> livre;
    public MaisonEdition() {
        this.livre = new ArrayList<Livre>();
    }
    public MaisonEdition(Integer idMaison) {
        this.livre = new ArrayList<Livre>();
        this.idMaison = idMaison;
    }
    public MaisonEdition(Integer idMaison, String nomMaison) {
        this.livre = new ArrayList<Livre>();
        this.idMaison = idMaison;
        this.nomMaison = nomMaison; }
    public Integer getIdMaison() { return idMaison; }
    public void setIdMaison(Integer idMaison) { this.idMaison = idMaison; }
    public String getNomMaison() { return nomMaison; }
    public void setNomMaison(String nomMaison) {
        this.nomMaison = nomMaison;}
    public List<Livre> getLivre() { return livre; }
    public void setLivre(List<Livre> livre) { this.livre = livre; }
    @Override
    public String toString() { return "livre_projet.MaisonEdition[
idMaison=" + idMaison + " ]";

```



ison_edition et livre).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:
s:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="Livre_projetPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>livre_projet.MaisonEdition</class>
    <class>livre_projet.Livre</class>
    <properties>

    <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:330
6/livre_JPA?zeroDateTimeBehavior=convertToNull"/>
      <property name="javax.persistence.jdbc.password" value=""/>

    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/
  >
    <property name="javax.persistence.jdbc.user" value="root"/>
  </properties>
</persistence-unit>
</persistence>
```

Références Biblio & Webographies

Bibliographie

http://nicolas.durand.perso.luminy.univ-amu.fr/pub/sar/Slides_05_Sockets.pdf

<http://sardes.inrialpes.fr/~krakowia/Enseignement/M1/Flips/6-Sock-RPC-RAR-M1.pdf>

<https://www.u-picardie.fr/~furst/docs/Thread.pdf>

<http://deptinfo.unice.fr/~grin/messupports/java/Thread6.pdf>

Book : Java reflection in action

Webographie

<http://docs.oracle.com/javase/tutorial/java/annotations/>

<http://www.google.com>

<http://java.developpez.com/cours>

http://fr.wikipedia.org/wiki/Wikip%C3%A9dia:Accueil_principal

<http://fr.slideshare.net/>

<http://openclassrooms.com/courses/introduction-aux-sockets-1>

<http://queinnec.perso.enseiht.fr/Ens/Chat/socket-java.html>

<http://openclassrooms.com/courses/la-programmation-systeme-en-c-sous-unix/les-threads-3>

<http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>

<http://alwin.developpez.com/tutorial/JavaThread/>