MEMO JAVA POO et developpement en couche

*Par Jean-Baptiste Cochinard*

Table des matières

[Base Java : quelques rappel 2](#_Toc71703266)

[La création de classes 3](#_Toc71703267)

[Les associations 3](#_Toc71703268)

[L’héritage 4](#_Toc71703269)

[Les classes abstraites, les méthodes abstraites et les interfaces 5](#_Toc71703270)

[Les exceptions 5](#_Toc71703271)

[Les énumérations 6](#_Toc71703272)

[La généricité 6](#_Toc71703273)

[Les collections 6](#_Toc71703274)

[Collection de liste : interface List 6](#_Toc71703275)

[Collection de type clé valeur : Interface Map 6](#_Toc71703276)

[Les classes générique 7](#_Toc71703277)

[Les méthodes générique 7](#_Toc71703278)

[L’architecture en couches 8](#_Toc71703279)

[Développement de la couche BO (Business Objects) 8](#_Toc71703280)

[Développement de la couche DAL 8](#_Toc71703281)

[Procedure stocké 9](#_Toc71703282)

[Transaction 10](#_Toc71703283)

[Externaliser une chaine de connexion 10](#_Toc71703284)

[Définir le DAO 11](#_Toc71703285)

[Utilisation de la couche DAL par la couche BLL 11](#_Toc71703286)

[Développement de la couche BLL 11](#_Toc71703287)

[Design pattern Singleton 11](#_Toc71703288)

[Couche IHM 12](#_Toc71703289)

[Création d’une premiere fenetre : 12](#_Toc71703290)

[Les composants graphique 12](#_Toc71703291)

[Placer les composants sur la JFrame : Les Layout 13](#_Toc71703292)

[Les composants 13](#_Toc71703293)

[Programmation evenementiel et design pattern Observer 13](#_Toc71703294)

# Base Java : quelques rappel

|  |  |
| --- | --- |
| **Out.println()**  **Out.printf()**  **Err.print()**  **Scanner**  **nextInt()**  **nextLine()**  **try…**  **catch…**  **finally**  **throws**  **throw**  *Ecrire dans un fichier*  *Lire un fichier* | *La sortie d’erreur peut par exemple écrire dans un fichier de log pour conserver un historique* |
| *Type par valeur et référence* |  |

# La création de classes

Une classe est responsable de ses données, c’est-à-dire que les attributs doivent restreindre leur visibilité pour que seul les méthodes définie puissent les consulter (getter) et les modifier (setter).

|  |  |
| --- | --- |
| **Package**  *Attribut*  *Visibilité*  ***Static***  *Méthode*  ***Static***  *Constructeur*  *Utilisation* |  |

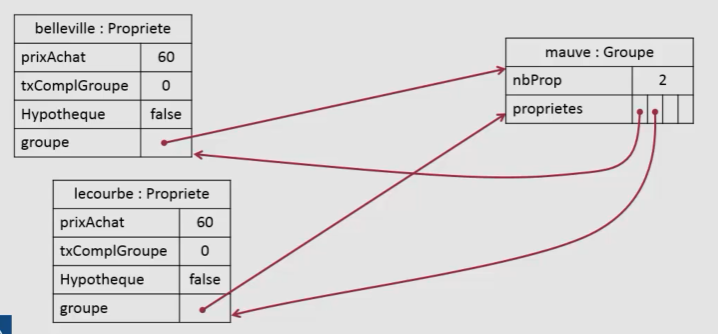
La **surcharge** : Il est possible d’avoir des méthodes de classes qui ont le même nom mais pas le même nombre de paramètre et/ou le même type.

# Les associations

Une association peut être de 2 types, unidirectionnel et bidirectionnel.

* Unidirectionnel veut dire que notre classe comportera elle-même des instances d’autres classe (ex String etc…) mais que les instances de String ne connaissent pas notre classe.
* Bidirectionnel veut dire qu’une propriété peut savoir a quel grouppe elle appartiens et vice-versa.

Attention lors d’association bidirectionnel, il faudra veiller à garder une cohérence des données à l’aide de vérification.



# L’héritage

L’héritage des classes permet de créer une classe a partir d’une autre et ainsi d’hérité des attributs et méthode. On peut dire qu’une classe hérite d’une autre si l’affirmation **B est un A** est vrai.

Toute classe hérite de la classe **Objet** et possède ses attributs et méthode comme .clone(), .equals(), .toString(), .getClass() etc)



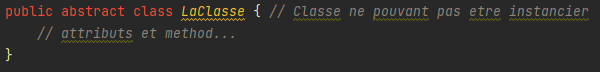
Il est possible de transtypé (convertir) un type de classe **C** en un type de classe **A** si celle-ci est une descendante

Le polymorphisme est la capacité d’une classe à utiliser les méthode les plus spécifique dans le cadre d’un appel par transtypage

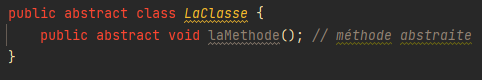
Ex : a est de type A et b est de type B (qui hérite de A) si on utilise une méthode tel que (A) b.method

# Les classes abstraites, les méthodes abstraites et les interfaces

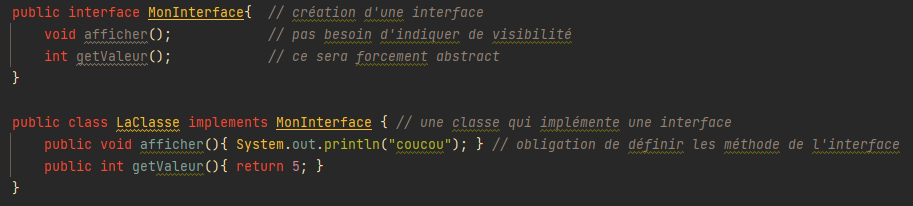
Une classe abstraites est une classe qui ne peut pas être instancier, elle sert à définir un parent sur lequel ont va **extends** d’autre classes



Dans cette classe abstraite on peut définir des méthodes abstraites, celle-ci va obliger les classes qui en hérite à donné un corps a cette méthode.



Une interface possède uniquement des méthodes abstraites et des constantes. Une classe contrairement a l’héritage peut avoir plusieurs interfaces, ont dit qu’on va l’implémenter. Les classes qui ont une interface sont obliger de définir les méthodes tout comme on le ferais avec une méthode abstraite.

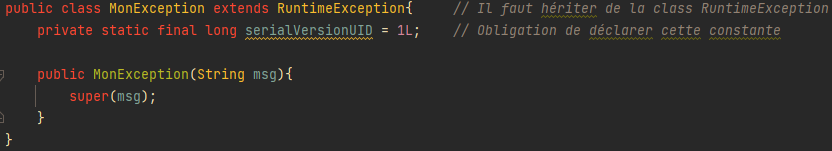


# Les exceptions

Une exception est lever lorsque le programme rencontre un problème, Ils existent 3 catégorie d’exception :

* les erreurs grave qui arrete le programme par exemple OutOfMemoryError, StackOverflowError etc…
* les exceptions non controlé sont des erreur de programmation detectées à l’execution, elles sont à corriger par le programmeur. Par exemple il y as le NullPointerException, ArrayIndexOutOfBoundsExeption etc…

**Exception non controlé personalisées :**



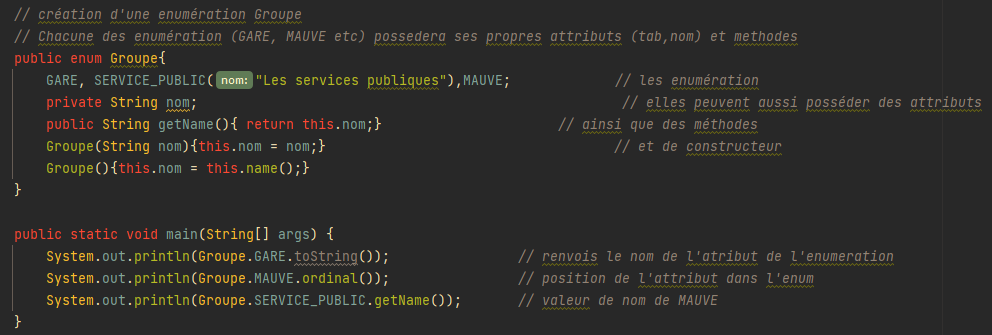
**Et pour lever l’exception :**



Les exception contrôlées, sont des erreur pour lesquel le system va tenter de régler le probleme en fesant remonter l’exeption, par exemple FileNotFoundException est une exception controlé

# Les énumérations

Une enumeration est une classe qui hérite de la classe Enum et Object.



# La généricité

## Les collections

Pour un meilleur maintenabilité du code il est préférable d’instancier une collection via la classe d’implémentation, ainsi il sera aisé de modifier le type de collections que l’ont utilise si on as besoin de modifier

**List<String> liste = new ArrayList<>() ;** // ainsi on pourra mettre une classe différente que ArrayList et utiliser les même méthodes

### Collection de liste : interface List

Une image contenant texte

Description générée automatiquement

### Collection de type clé valeur : Interface Map

Une image contenant texte

Description générée automatiquement

## Les classes générique

Une **classe générique** permet de créer une classe dont au moins une propriété est d’un type de classe pas encore connue, on dit que ce paramètre est de **type T**. Ensuite lors que l’instanciation de notre **classe générique** on donne en paramètre la classe que l’ont souhaité utiliser, ont dit alors que cette classe est une **classe de réalisation**.

Ce concept est ce qu’il se passe pour l’arrayList, qui est une classe générique, dont le parametre T est définie lors que l’instantiation.

Les types génériques *ne peuvent pas être remplacés par un type de base* (int, char, long etc) en revanche on peut utiliser la classe **wrapper** correspondante (Integer, Long, Character,Byte etc)

Une image contenant texte, extérieur

Description générée automatiquement

Création d’une classe generique

Une image contenant texte

Description générée automatiquement

Et ensuite pour instancier notre classe : **Test<String> t = new Test<>() ;** // il faut définir le type T

## Les méthodes générique

Une méthode générique tout comme la classe permet de pouvoir utiliser un paramètre de type T (donc non connu d’avance) dans une méthode.

Une image contenant texte

Description générée automatiquement

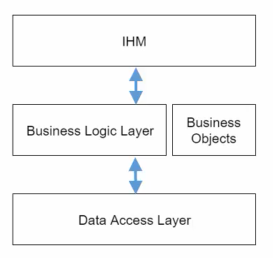
On peut également faire des contraintes sur le type T, par exemple je veux que mon type implémente une interface Comparable

Une image contenant texte

Description générée automatiquement

On peut déclarer plusieurs extends en respectant le fait que : le premier extends soit une classe si on souhaite en définir une, puis les interfaces (autant que l’ont souhaite) séparer par un &

# L’architecture en couches

Les couches d’une application sont les différents processus qui sont mis en œuvre afin de fournir une application a l’utilisateur. Cette architecture peut se décomposer en 3 couches :

* **HIM** : la présentation (ce que vois l’utilisateur), et la saisie des données,
* **Business logic layer (BLL)** : la logique métier (tout ce qui se rapporte au calcule, traites les données) et Business Objects (**BO**) le modèle de données
* Data Access Layer (**DAL**): persistance des données (SQL, DD par exemple)

Chaque couche logiciel est un composant pouvant communiquer avec ses couches voisines mais uniquement celle-ci. Une couche permet de masquer les détail l’implémentation de la couche supérieur : par exemple l’IHM demande a récupérer des utilisateurs sans avoir connaissance de comment cela est récupéré. Cela permet par exemple de modifier la base de données (migration vers un autre) en ne modifiant que la couche concerné).

L’implémentation de l’architecture en couches peut se réaliser de plusieurs manières par exemple on peut :

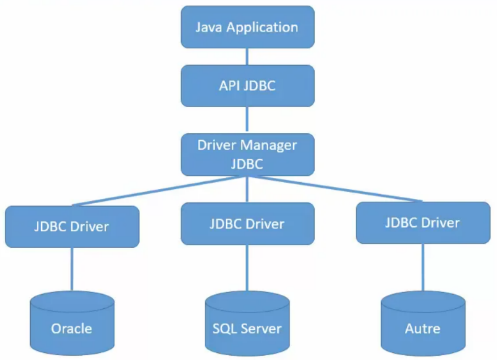
* Créer un package par couche (ihm, bo, bll, dal), cela créer une dépendant forte entre les couches car écrit en dur
* Créer un projet par couche (ihm,bo,bll,dal). Il faut modifier le build path pour trouver les classes qui sont autre projet
* En utilisant une archive .jar qui sera en faite un projet zippé que l’on glisse dans les ressources pour l’utiliser il faut également déclarer le jar dans le build path et l’ajouter en tant que librairy.

# Développement de la couche BO (Business Objects)

La business objects (**BO**) fait partie de la couche métier, elles à été séparer de la business logic layer (**BLL**) pour facilité son utilisation dans les couches IHM et DAL

La BO utilisent des classes simple, dites **POJO** (Plain Old Java Objects) qui sont des objets lié a aucune dépendance ou framework, on n’y utilise que du code et des classe natif de JAVA.

# Développement de la couche DAL

La couche DAL est responsable de la sauvegarde des données, sur des supports tel que fichier ou base de données. DAL est aussi nommée couche de persistance des données.

Pour communiquer avec une base de données on va utiliser une API nommée JDBC. Chaque fournisseur de base de données (Oracle, Microsoft, MySQL) a créé des drivers permettant d’implémenter leur solutions.

Le Java de base ne founit que les Interfaces de JDBC, leur implémentations (driver) est définie par les fournisseurs et est a telechargé et a ajouter en tant que librairie externe.

La connexion à une base de données de déroule en 2 étapes :

* ***Le chargement du pilote JDBC par appel de la méthode Class.forName(url) ; ou url est l’adresse du jar***

Une image contenant texte

Description générée automatiquement

* ***L’établissement de la connexion en créant un objet de l’interface Connection***

Une image contenant texte

Description générée automatiquement

* ***La création de Statement qui permet d’interagir avec la BDD***

Une image contenant texte

Description générée automatiquement

***Requête préparer***

Une image contenant texte, capture d’écran, noir, fermer

Description générée automatiquement

## Procedure stocké

Il faut créer un objet de type CallableStatement dans lequel on fournis une requête sql sous forme de chaine de caractere

Une image contenant texte

Description générée automatiquement

## Transaction

Par defaut on est en auto-commit (chaque instruction est validé des qu’elle sont envoyé au serveur) nous pouvons modifier ce comportement

Une image contenant texte

Description générée automatiquement

## Externaliser une chaine de connexion

Il est préférable de définir les paramètre d’accès a notre base de données dans un fichier externe afin de ne pas devoir recompiler les informations à chaque modification.

Pour cela on créer une classe que l’ont peut nommé Settings dans laquelle on va définir une propriété **Properties**

**Cela se fait en 3 étapes :**

1-création d’un fichier en .properties ou seront contenue nos propriétés

Une image contenant texte

Description générée automatiquement

2- création d’un classe Settings permettant l’accès aux propriétés Une image contenant texte

Description générée automatiquement

3- Utilisation

Une image contenant texte

Description générée automatiquement

## Définir le DAO

Le Data Access Object est le composant principale de la couche DAL, il contient le code qui va vraiment communiquer avec le serveur. Par convention on nomme le DAO par le domaine métier et la tech utiliser :

## Utilisation de la couche DAL par la couche BLL

Pour ne pas dépendre d’une implémentation spécifique d’accès aux données, il faut encapsuler l’instanciation des instances de la couche BLL qui vont communiquer avec la DAL, il faut passer par une « factory » c’est ce que l’on appel le **design pattern DAO**. (ci-contre)

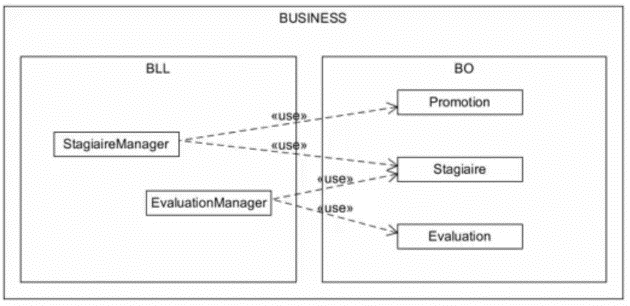
Notre BLL communiquera avec la DAOFactory qui renverra une instance de l’implémentation.

Pour mettre en place le design pattern DAO il faut :

* Créer une classe **Settings** (avec le fichier qui va avec) qui contiendra l’accès au paramètre de connexion
* Créer une classe **JdbcTools** qui contiendra le code pour accéder à une connexion et fermer la connexion (cette classe utilise la classe Settings)
* Créer une classe de type **ClasseDAOJdbcImpl** dans la couche DAL qui s’occupe de faire les requêtes
* Créer une interface **ClasseDAO** dans la couche DAL qui reprend tous les entêtes de méthodes de **ClasseDAOJdbcImpl**
* Créer une classe Factory qui renverra une instance de notre **ClasseDAOJdbcImpl** via un appel depuis la couche supérieur (BLL) qui instancie via l’interface.

Exemple depuis la BLL il n’y as plus qu’a faire : **ClasseDAO daoMaClasse = Factory.getClasseDAO() ;**

# Développement de la couche BLL

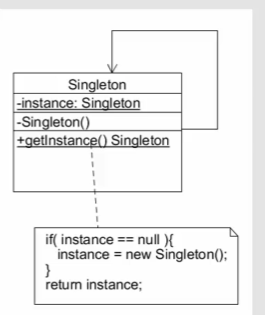
C’est la couche principale de notre programme, c’est la ou est stocker tout les traitement, c’est la couche dite métier. Elle communique avec la DAL et l’IHM, cette couche doit être indépendante des autres couches et ainsi il sera plus simple de réutiliser cette couche dans un autre programme.

La logique métier de la couche BLL est repartie dans diffèrent composants, par domaine métier appeler manager, ci-contre nous avons 2 manager qui sont StagiaireManager  et EvaluationManager.

Pour fonctionner les manager utilisent les BO qui sont les objets principaux de la logique métier. Un Manager est comme un chef d’orchestre qui fait le lien entre les couches IHM et DAL.

L’instanciation via le mot clé new est couteux, c’est pourquoi il existe une méthode, un pattern… (suspens)

## Design pattern Singleton

Le design pattern permet de garantir l’unicité d’un objet dans une application. L’instance unique sera stockée dans un attribut statique du type du Singleton.

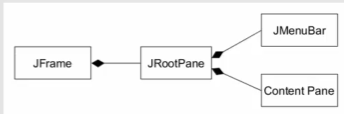
Une image contenant texte

Description générée automatiquement La création de l’instance est gérer par getInstance() qui retourne la référence, si celle-ci est null on l’a créer sinon on renvois notre attribut, la méthode est statique (il va de soi 😉)

*Exemple d’implémentation à droite.*

# Couche IHM

La couche est responsable de présenter les informations et de capturer les saisies de l’utilisateurs. On utilise une API nommé Swing afin de créer une interface IHM. Swing gère des composant graphique (bouton, champs, tables, panneaux etc) et le Look and feel (accessibilité handicap), il y a aussi une bibliothèque de dessin en 2D.

Une **IHM** en **swing** contient un élément principal, dit conteneur de haut niveau, qui contiens tout les autres composants graphique (comme **JFrame** et **JPane**l). Un composant de haut niveau est composé de 3 structures, le **Root Pane** sur lequel on interagit relativement peu, le **JMenuBar** qui est la barre de menu et le C**ontentPane** (contenu de la fenêtre) sur lequel on viendra poser nos éléments graphiques.

Swing propose 3 conteneurs principaux, la **JFrame** (écran principal avec bouton fermeture etc), la **JDialog** (écran secondaire, dépendant d’un **JFrame**) et un **JApplet** (technologie déprécié).

## Création d’une premiere fenetre :

1er étape on définie une classe qui hérite de **JFrame**

Une image contenant texte

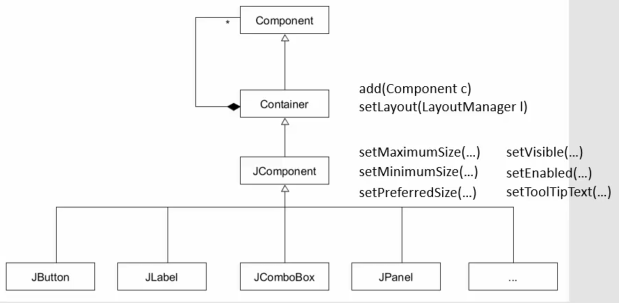
Description générée automatiquement

2em étape on créer un thread spécifique dans lequel exécuter notre fenêtre

Une image contenant texte

Description générée automatiquement

## Les composants graphique

Tous les composants héritent de **JComponent** et possède des méthodes tel que (setMaximumSize(), setVisible() etc). **JCompenent** est lui-même un Container, ce qui veut dire que certains composants peuvent eux même avoir des composants (ex : **JPanel**)

## Placer les composants sur la JFrame : Les Layout

La classe **JFrame** fournis un attribut **ContentPane** de type Container, on peut redéfinir ce **ContentPane** avec la méthode **setContentPane()** dans notre classe **JFrame**.

Ensuite c’est dans ce **contentPane** que l’on ajoutera les élément graphique via la méthode **add()**. Le conteneur dispose d’un Layout (une mise en page automatique des éléments). Chaque conteneur à un layout associé par défaut. La gestion du layout se fait par deux méthode principalement, **setLayout(LayoutManager layout)** et **add(Component c)**

Les différent layout :

* **BorderLayout** (organise en la page en 5 zones (PAGE\_START, PAGE\_END, LINE\_END, LINE\_START, CENTER)
* **BoxLayout** : range-les compostant en une colonne ou ligne
* **CardLayout** permet de superposer les dispositions
* **FlowLayout** dispose les éléments en rangé en passant a la ligne quand nécessaire
* **GridBagLayout** dispose les éléments en tableau avec la possibilité d’étendre les éléments à n ligne/colonne
* **GridLayout** dispose les éléments en tableau de cellule égale

## Les composants

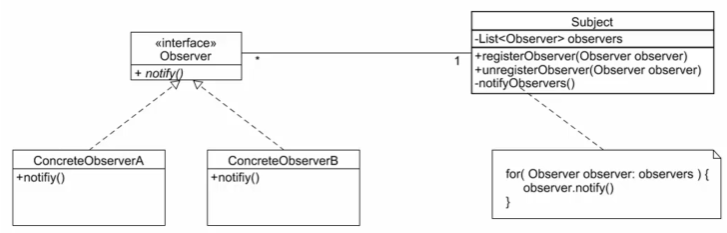
**Une image contenant texte

Description générée automatiquement *Ci-contre les principaux composants de Swing.=>*

*Il est conseiller d’ajouter tous les boutons dans un* ***JPanel*** *spécifique*

<=

## Programmation evenementiel et design pattern Observer



Permet de mettre à jour en temps réel. Chaque ajout/suppression/modification on notifira les vues sur l’interface Observer

Une image contenant texte

Description générée automatiquement**Ajout d’un listener sur un composant via une classe anonyme**

**Mise en pratique :**

Une image contenant texte

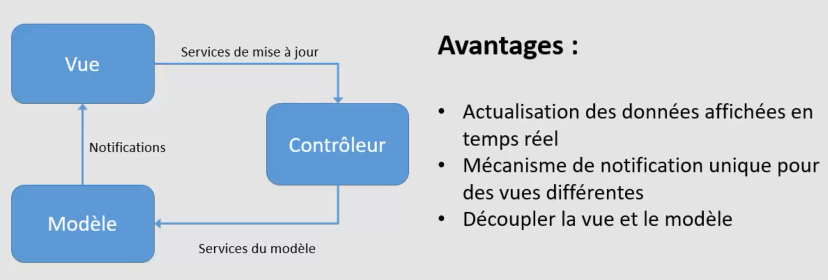
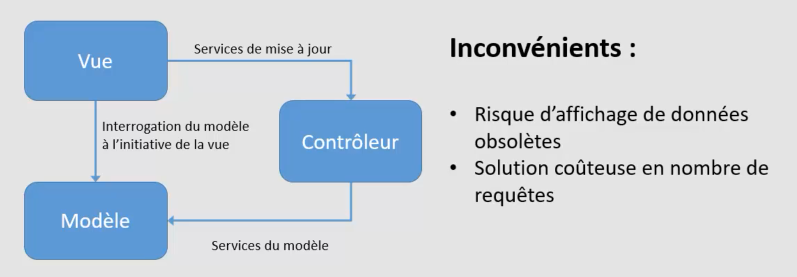
Description générée automatiquement

# L’architecture Modele Vue Controleur (MVC)

L’architecure MVC permet de segmenter le code en 3 grande partie :

// todo dif avec 3tiers

## Mise à jour de la vue peut être faite de deux manière, la push est la préférable

**PULL** **PUSH**