

---

# Formation

**Environnement**

**ECLIPSE ( IDE java )**

---

## Table des matières

<b>I - Eclipse : introduction, prise en main.....</b>	<b>4</b>
1. Présentation de l'IDE eclipse.....	4
2. Concepts de base.....	4
3. Perspectives.....	5
4. Configurations fondamentales [eclipse , projets].....	6
<b>II - Developpement Java avec eclipse.....</b>	<b>8</b>
1. Programmation java avec eclipse.....	8
<b>III - Lombok et eclipse.....</b>	<b>15</b>
1. Lombok.....	15
<b>IV - Debug eclipse.....</b>	<b>17</b>
1. Debug avec eclipse.....	17
2. Debug web (remote debug).....	22
<b>V - Maven et eclipse.....</b>	<b>24</b>
1. Présentation de maven.....	24
2. Cycle de build , tâches et plugins.....	25
3. Structure d'un projet et conventions.....	28
4. Lien entre maven et eclipse (m2e).....	41
<b>VI - JUnit et eclipse.....</b>	<b>47</b>
1. Les principes des tests unitaires.....	47
2. Tests unitaires avec JUnit (3 ou 4).....	49
3. JUnit et Maven.....	55
<b>VII - GIT et eclipse.....</b>	<b>56</b>
1. Présentation de GIT.....	56
2. Configuration locale de GIT.....	57
3. Principales commandes de GIT (en mode local).....	57
4. Commandes de GIT pour le mode distant.....	59
5. Gestion des branches avec GIT.....	60

---

6. Gérer plusieurs référentiels distants.....	62
7. Plugin eclipse pour GIT (EGIT).....	62

<b>VIII - Data-explorer et JDBC , Hibernate/JPA.....</b>	<b>64</b>
--	-----------

1. Eclipse et les bases de données.....	64
---	----

<b>IX - Configuration eclipse (plugin, ....).....</b>	<b>71</b>
---	-----------

1. Configurations (avancées) eclipse.....	71
---	----

<b>X - Annexe – Liste des Travaux Pratiques.....</b>	<b>73</b>
--	-----------

1. TP "prise en main d'eclipse".....	73
2. Tp "Developpement java avec eclipse".....	73
3. Tp "debug".....	73
4. Tp "maven et eclipse".....	73
5. Tp "JUnit et eclipse".....	73

# I - Eclipse : introduction, prise en main

## 1. Présentation de l'IDE eclipse

Eclipse est un **IDE** (*Environnement de Développement Intégré / Integrated Development Env.*).

**Cet environnement intègre tous les outils nécessaires au développement d'une application java :**

- éditeur de code (avec assistants : auto-complétion, détection des erreurs , ...)
- compilation (automatique et/ou contrôlée)
- debugger (mode pas à pas , ...)
- lien avec le système de gestion de version (SVN ou GIT ou ...)
- lien avec le gestionnaire de projet "maven" (téléchargement des dépendances , packaging , ... )
- lancement de tests unitaires (JUnit)
- assistants spécialisés (data explorer , ...)
- liaison avec un serveur JEE (ex : tomcat ou Jboss) pour déployer et tester une application "web" java.
- ....

L'IDE **Eclipse** est structuré de façon **modulaire** en étant bâti sur une architecture à base de **plugins**.

L'IDE Eclipse est **open-source** et les premières versions ont bénéficié d'une forte contribution de la part d'IBM .

Il y a quelques années , le cœur d'eclipse a été profondément remanié (passage des versions 3.x aux versions 4.x) . Il faut au minimum la version 4.3 / Kepler pour disposer d'un environnement 4.x stable.

Les versions récentes sont **4.4 / Luna** , **4.5 / Mars** et **4.6 / Neon** .

Chaque année, une nouvelle version sort en été et des correctifs \_SR1 , \_SR2 viennent parfaire celle-ci en automne et en hiver.

## 2. Concepts de base

<i>Eléments d'eclipse</i>	<i>Fonctionnalités</i>	<i>Particularités</i>
<b>Installation (logiciel eclipse)</b>	Code java du logiciel eclipse (eclipse/eclipse.exe , ...)	Code segmenté en une multitude de plugins
<b>Espace de travail (workspace)</b>	Répertoire de travail avec une	Le sous répertoire caché

	configuration spécifique et une multitude de projets potentiels	".metadata" comporte toute la configuration (potentiellement assez énorme) d'un workspace
<b>Projets</b>	Projets logiciels (application ou parties d'application à développer)	Chaque projet comporte sa propre configuration (qui complète celle du workspace)
<b>Natures</b> d'un projet	Types éventuellement complémentaires d'un projet (ex : <b>java</b> , <b>maven2</b> , ...)	Dans <natures><nature> du fichier caché <b>.project</b>
<b>Ressources</b> d'un projet	Arborescence de répertoires et de fichiers (code , images , configurations, ...)	Un " <b>Refresh</b> " est à déclencher en cas de désynchronisation avec le contenu du file-system
<b>Facettes</b> technologiques	Liste des technologies (api , ...) utilisées sur un projet (ex : java8 + jsf 2.2 + jpa 2.1 + .... )	Les facettes technologiques ont une grande incidence sur les éléments visibles ou pas des menus de certaines perspectives
<b>Perspectives</b>	Organisations interchangeables de jeux de vues/fenêtres au niveau de l'IDE (ex : Java/EE , Debug , ...)	
Ensemble de projets ( <b>WorkingSet</b> )	Paquet cohérent de projets que l'on peut gérer globalement (ouverture , fermeture , ...)	Les "workingSet" sont facultatifs et ne sont utiles que s'il faut gérer/organiser un grand nombre de projets

Un projet ne peut être géré qu'à l'intérieur d'un espace de travail.

Par défaut , le répertoire d'un projet est contenu dans le répertoire du workspace. Il est cependant possible de placer/référencer le répertoire correspondant à un projet au dehors du répertoire du workspace (après git clone par exemple) .

Dans certains cas (heureusement très rares) , les fichiers de configuration d'un workspace peuvent devenir instables/incohérents et eclipse peut refuser de redémarrer. Un remède (assez radical) consiste alors à supprimer le répertoire caché ".metadata" du workspace , de redémarrer eclipse et de reconfigurer le workspace (configurations, réimportation des projets) .

Souvent une éventuelle désactivation et une activation d'une facette technologique permet de générer certains fichiers de configuration attendus par cette technologie (avec les bonnes entêtes XML/XSD) .

Par exemple l'activation de la facette "JSF 2 .x" permet générer (entre autres) le fichier WEB-INF/faces-config.xml .

### 3. Perspectives

L'environnement **eclipse** a la particularité d'être basée sur la notion de perspectives.

Une **perspective** est une **vision particulière du projet (sous un certain angle)**:

- Resources = Visualisation et navigation par fichiers et répertoire
- Java = Visualisation par packages et classes java
- JEE = Visualisation par modules et par composants "métier" (ejb, ...).
- Debug = Environnement pour effectuer les tests (avec console, ...).
- ...

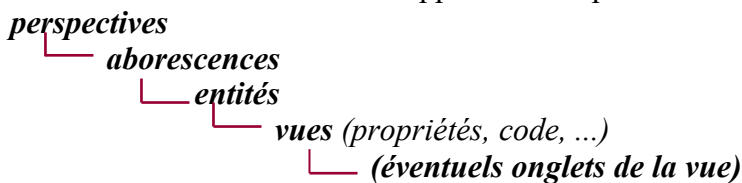
On bascule d'une perspective à l'autre via des minis onglets situés par défaut sur le coté gauche.



Chacune de ces **perspectives** correspond à un (sous) **environnement de travail virtuel** et est composée d'un ensemble de *fenêtres* (appelées *Vues*).

La **principale vue d'une perspective** est généralement une **arborescence** permettant de naviguer à l'intérieur d'un ensemble d'entités pour en sélectionner une et ainsi afficher ses détails dans différentes vues secondaires.

==> L'environnement de développement comporte donc *n niveaux*:



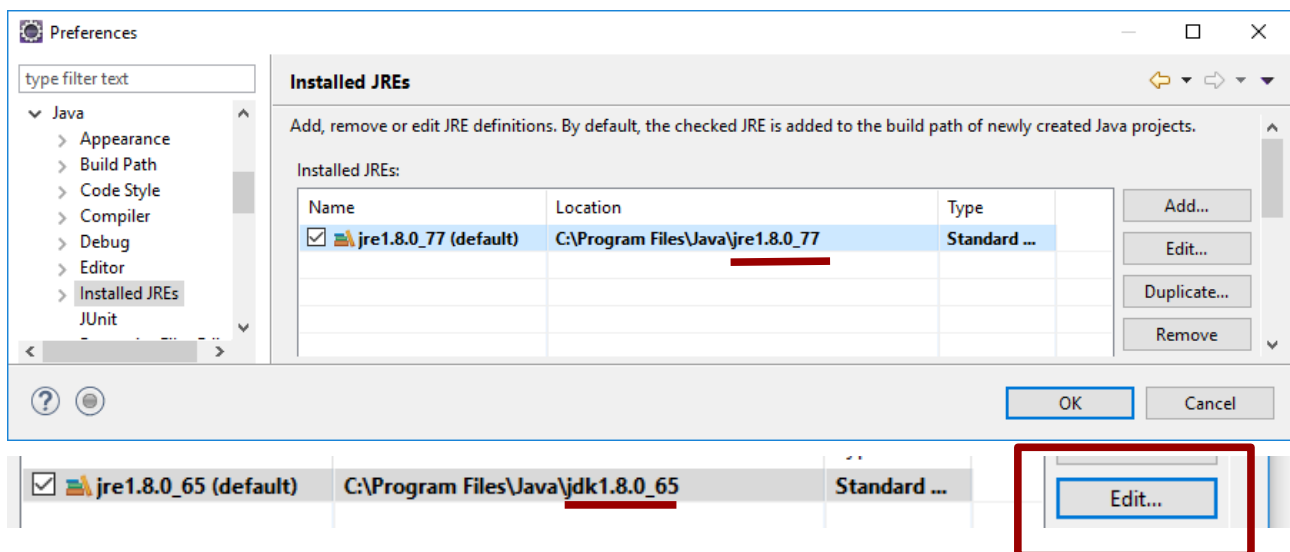
## 4. Configurations fondamentales [eclipse , projets]

### 4.1. Configurations générales / globales pour un workspace

Menu principal pour configurer globalement l'ensemble de l'environnement de développement :

==> **Window / Preferences**

Paramétrage clef: sous menu "**Java / install JRE**" ==> chemin(s) d'accès au(x) JDK(s)



**NB**: en reparamétrant le "default JRE" de façon à pointer vers le répertoire "jdk\_..." plutôt

que vers le répertoire "jre\_..." , on améliore énormément le comportement d'eclipse : les assistants fonctionnent mieux (le code des méthodes générées comporte des paramètres ayant des noms précis au lieu de "arg0" , "arg1" , ... "argN" , on peut lancer des "build maven" , ... ) .

Menu à retenir pour faire apparaître une vue manquante dans la perspective courante:

==> **Window / Show View**

## II - Développement Java avec eclipse

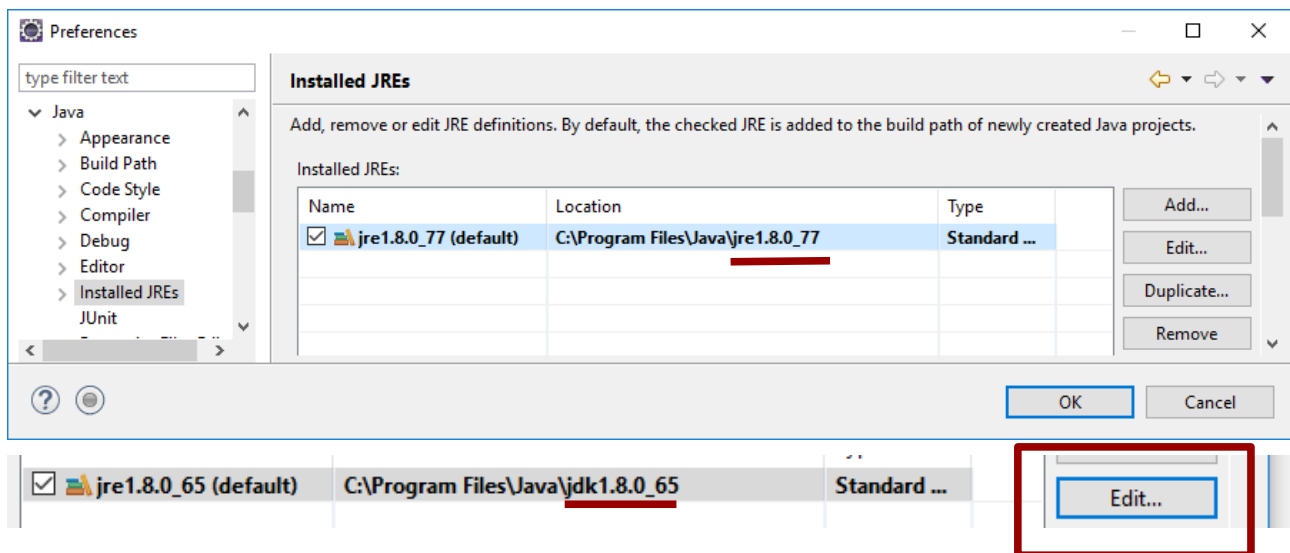
### 1. Programmation java avec eclipse

#### 1.1. Rappel du paramétrage fondamental

Menu principal pour configurer globalement l'ensemble de l'environnement de développement :

==> **Window / Preferences**

Paramétrage clef: sous menu "**Java / install JRE**" ==> chemin(s) d'accès au(x) JDK(s)



#### 1.2. Avant propos important

Historiquement, les premières versions d'eclipse ont été inventées avant l'existence de "maven".

Un projet java/jee (sans maven) était (et peut potentiellement encore être) uniquement géré par eclipse.

Ces anciens types de projets existent encore dans eclipse :

- **java project** (application java simple , autonome ou bien librairie)
- **dynamic web project** (application web avec servlet/jsp et/ou JSF ou ... )
- **ejb project**
- **entreprise application project** (point central d'une application jee complète / EAR référençant des sous projets "web" , "ejb", ... )
- ...



Aujourd'hui une très grande majorité des projets java/jee d'entreprise sont basés sur la technologie "**maven**" et les types de projets évoqués plus haut (purement "eclipse") sont devenus petit à petit obsolètes . C'est dans la partie <packaging> du fichier "maven" **pom.xml** que l'on précise le type de projet ("**jar**" , "**war**" , .... )

Peut être que dans quelques années, un grand nombre de projets seront gérés via la technologie moderne "gradle" plutôt que "maven" et l'on utilisera alors principalement le plugin et projets "gradle" .

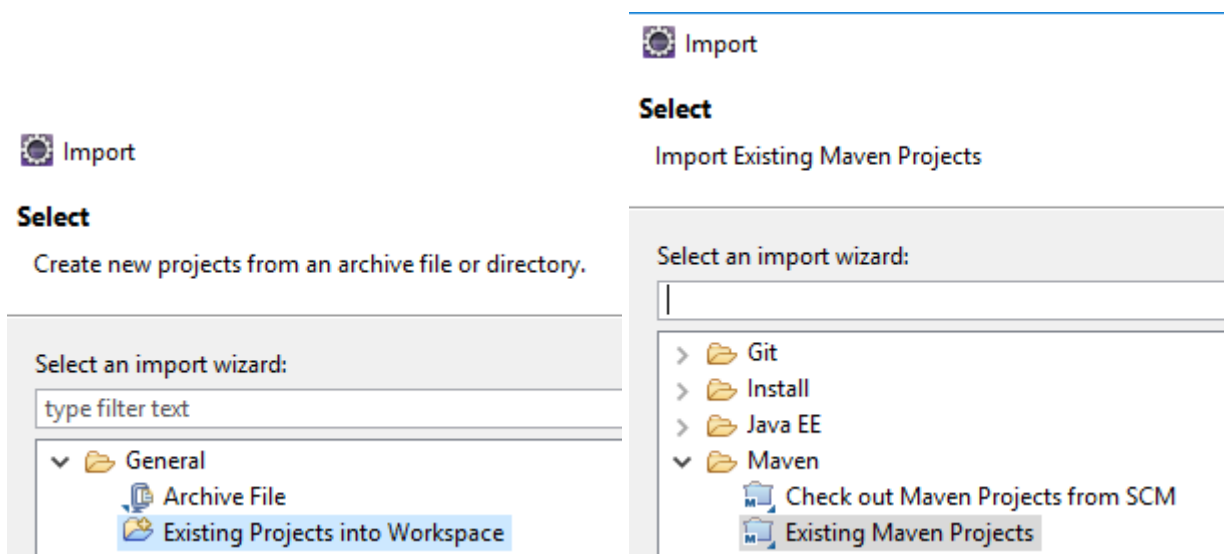
Il est donc très important de distinguer:

- **projets purement eclipse** ("java" , "dynamic web" , "ejb" , ....)
- **projets "maven"** gérables (entre autres) par eclipse
- autres types de projets ("gradle" , ...)

Principale différence :

Pour importer dans un workspace eclipse un **projet purement eclipse** ("java" , "dynamic web" , ...) il faut activer le menu "**import / General / existing projects into workspace**"

alors que pour importer dans un workspace eclipse un **projet "maven"** il faut activer le menu "**import / maven / existing maven project**" :



### 1.3. Compilation et lancement d'un programme:

Pour recompiler le projet courant:

==> **Project / Build**

ou simplement enregistrer le fichier source si *project / build automatically (\*)*

**(\*) par défaut , l'environnement eclipse compile automatiquement un fichier source ".java"**

dès que l'on sauvegarde celui-ci (en cliquant sur l'icône disquette) . Selon le type de projet , le résultat de la compilation (fichier ".class") sera placé dans le répertoire "bin" , "build" ou autre .

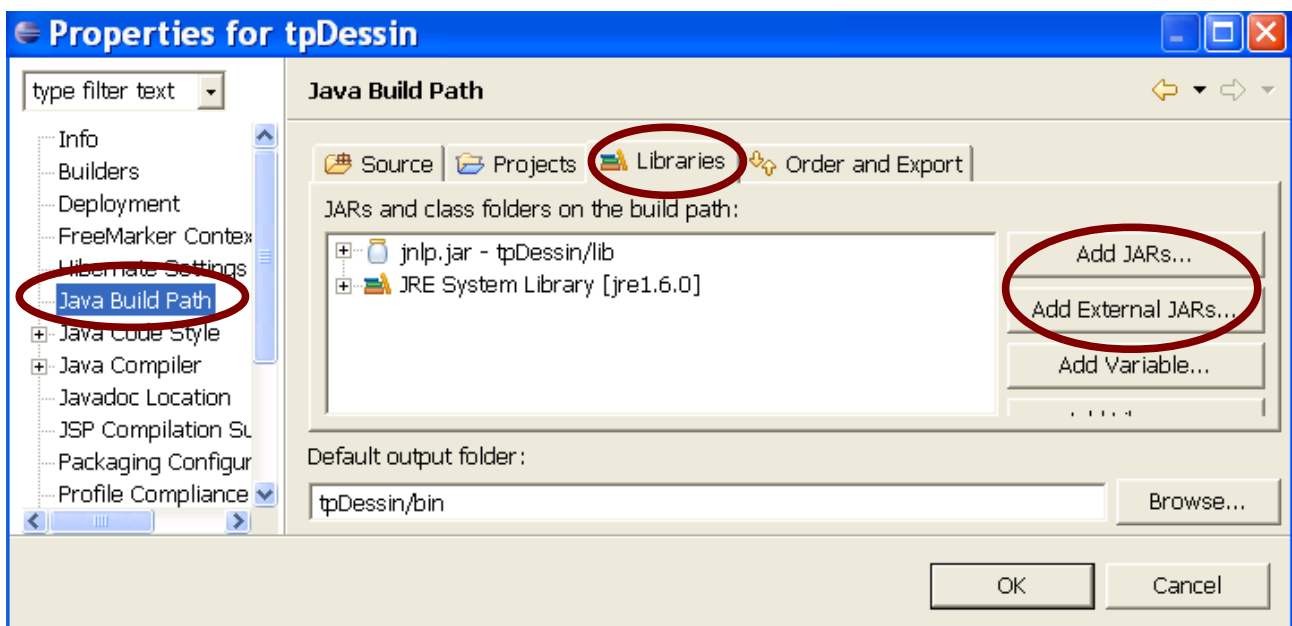
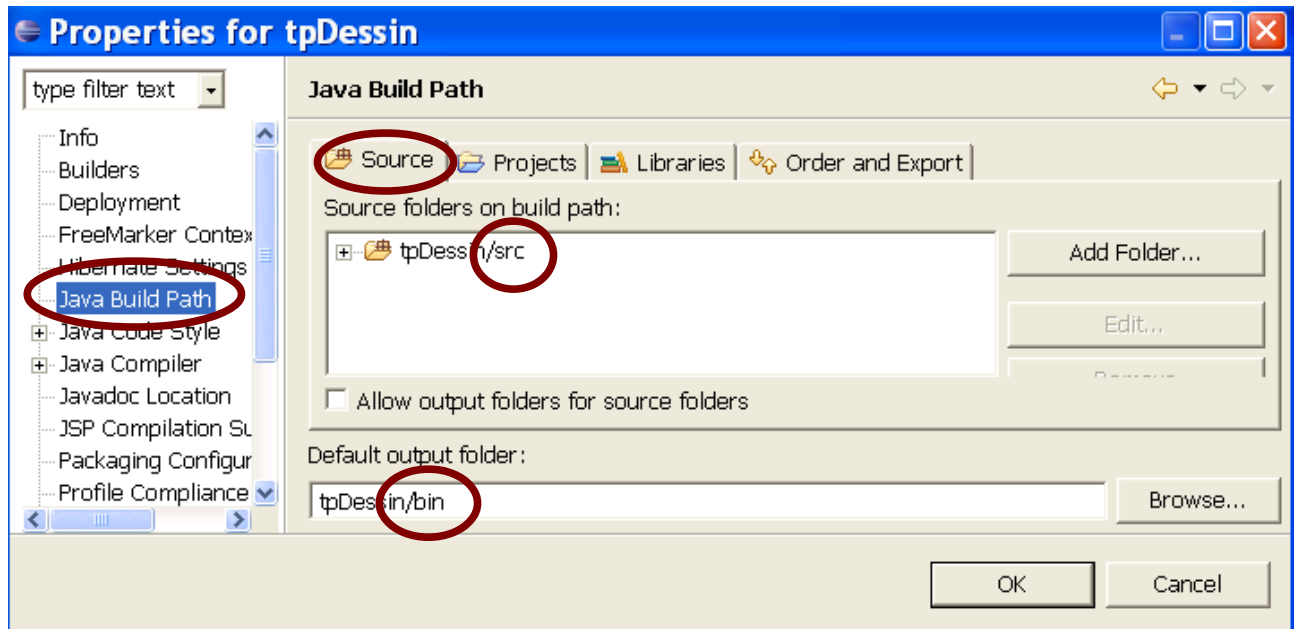
==> De temps en temps "**Refresh**" + **Project / clean** est utile pour forcer à recompiler tout le projet (lorsque le mode automatique optimise trop)

**Pour lancer le programme courant:**

- 1) sélectionner la classe comportant la méthode **main(...)**
- 2) click droit / **Run as Java Application** [ + paramétrages fin dans le menu Run/Run ...]

## 1.4. Configurations spécifiques à un projet "purement eclipse"

- 1) Sélectionner un projet
- 2) click droit / Properties



NB :

- dans un projet "java" ordinaire , il faut préciser tous les ".jar" à ajouter au classpath
- dans un projet "dynamic web" , tous les ".jar" placés dans WEB-INF/lib sont automatiquement déjà intégrés au classpath
- dans le cas d'un projet maven , c'est eclipse qui doit s'adapter à la configuration maven et il vaut mieux configurer le fichier pom.xml et activer le menu "maven / update"

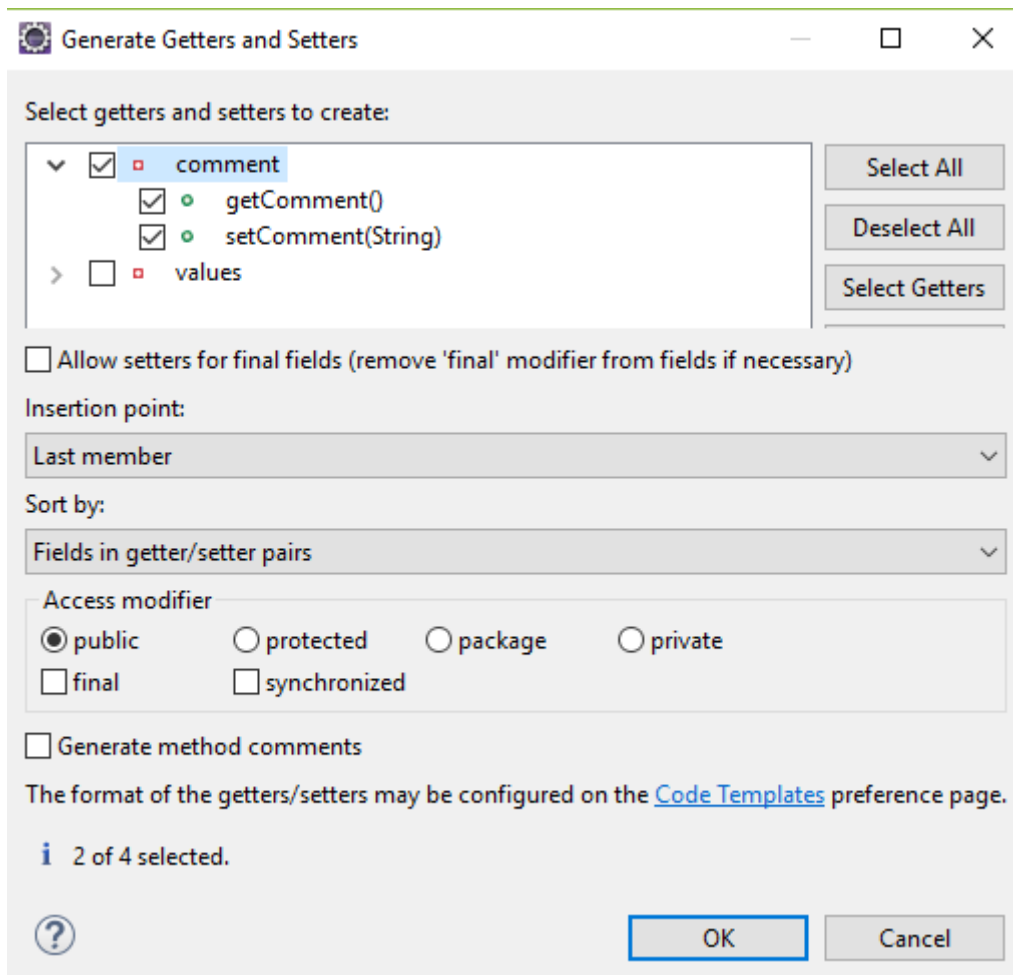
## 1.5. Edition du code source

- 1) se placer dans un éditeur de code java
- 2) click droit / **Source** / ....

**Generate Getter/Setter** ==> génère les méthodes public getXxx() et setXxx(...) en fonction des attributs privés "xxx"

**Organize imports (Ctrl-Shift-O)** ==> ajoute les "import ....;" manquants et retire les "import ....." inutiles

....



**Principaux assistants accessibles via le menu contextuel "source ..." :**

<i>Entrée du menu source</i>	<i>Raccourci clavier</i>	<i>Fonctionnalité</i>
Toggle comment	Ctrl+/ 	Ajoute ou retire des // au début des lignes sélectionnées pour les mettre ou pas en commentaire
Correct indentation	Ctrl+I	Corrige l'indentation de la ligne sélectionnée
Format	Ctrl+Shift+F	Formate correctement tout le code d'un fichier (bonne indentation , ...)
Organize import	Ctrl+Shift+O	Ajoute imports manquants , retire les imports inutiles .
Override/implements method		Redéfinit une méthode héritée (pour écrire un code plus spécifique) ou bien implémente une méthode imposée par une interface
Generate Getters and Setters		Génère les getXxx() , setXxx(...) pour les attributs privés xxx
Generate toString()		Génère le code de la méthode toString() en fonction des attributs sélectionnés
Generate hashCode() and equals()		Génère le code des méthodes hashCode() et equals() - pratique pour bien gérer de façon optimisée les éléments dans un "Set"
Generate Constructor using fields		Génère le code d'un constructeur ayant en paramètre 0 ou n argument(s) en fonction des champs sélectionnés
Generate Constructors from superclass		Génère des constructeurs qui ré-appellent ceux de la classe héritée (via super(...))

## 1.6. Auto-complétion au niveau de l'écriture du code java

La combinaison de touches "**Ctrl-Espace**" permet d'activer des propositions d'auto-complétion (en fonction par exemple du préfixe objet ou bien des premiers caractères saisis ).

Ensuite ,

- La touche "**entrée**" (ou bien un click sur une des propositions) permet d'effectuer une sélection de complétion .
- La touche "**echap**" permet de revenir dans un mode sans auto-complétion.

**NB:** L'**auto-complétion** déclenchée par "**Ctrl+Espace**" est **intelligente** et **contextuelle** et fonctionne dans les différents cas suivants :

- après "**this.**" pour sélectionner un attribut ou une méthode

- après "**NomClasse.**" pour sélectionner une méthode statique  
(exemple : `JOptionPane.showMessageDialog(...)` )
- après "**NomInterface.**" pour sélectionner une constante possible  
(exemple : `Connection.TRANSACTION_READ_COMMITTED` )
- après **@** pour sélectionner une annotation possible (ex : `@Id` , `@Entity` , ....)
- à l'intérieur des parenthèses d'une annotation pour sélectionner un des paramètres possibles de l'annotation (ex : `@WebService(endpointInterface)` )
- après **nomParametre=** pour éventuellement sélectionner une valeur possible (énumérée) d'un paramètre lorsque c'est possible

### 1.7. Restructuration du code / refactoring

1. Sélectionner un package ou une classe ou un membre (attribut/méthode)
2. click droit **Refactor/Rename ...** ==> renomme non seulement la chose sélectionnée mais aussi tous les autres morceaux de code du projet qui y font référence .

Autre façon efficace pour réorganiser le code : effectuer des "glisser poser" d'un package à un autre . (pour déplacer une classe par exemple) .

## III - Lombok et eclipse

### 1. Lombok

#### 1.1. Fonctionnalités de lombok et principes de fonctionnement

Beaucoup de frameworks java nécessitent que l'on programme des classes java avec tout un tas de choses classiques mais répétitives ( getter/setter , constructeurs , méthode toString() , .... ).

**De façon à gagner du temps dans la programmation et à aérer le code source on peut éventuellement s'appuyer sur des annotations telles que @Getter , @Setter de la technologie lombok de façon à ce que les méthodes getXxx() et setXxx() soient automatiquement générées avec le code basique par défaut . Il est heureusement toujours possible de personnaliser quelques getters/setters .**

Depuis le jdk 1.6 (et Pluggable Annotation Processing API (JSR 269)) , les annotations de rétention "SOURCE" , telles de celles de lombok sont détectées dans le code source à compiler et traitées automatiquement durant la phase de compilation à partir du moment où le classpath comporte les ".jar" des traitements associés aux annotations. Il n'est donc plus nécessaire de lancer explicitement l'utilitaire apt du jdk 1.5 .

Lorsque la compilation du code est gérée par la technologie maven, les annotations de lombok (@Getter , @Setter , ...) sont interprétées et traitées automatiquement (si la dépendance maven "lombok" est présente dans pom.xml ) .

#### 1.2. Dépendances "maven" pour lombok

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.16.10</version> <!-- or via spring boot version -->
</dependency>
```

#### 1.3. Installation de l'agent lombok dans un IDE (ex : eclipse)

Dans le cas particulier d'une compilation lancée par eclipse , une synchronisation doit être établie entre le code source et le code compilé (dès chaque enregistrement du code source modifié).

De façon à ce que les annotations de lombok soient bien prises en compte par l'IDE eclipse (ou autre), une configuration doit être effectuée. Celle-ci peut s'effectuer de la façon suivante :

`~/m2/repository/org/projectlombok/lombok/1.16.10`

`$ java -jar lombok-1.16.10.jar` (ou double click sous windows)



→ un redémarrage de l'IDE (ex : eclipse) est nécessaire.

## 1.4. Paramétrages (@Getter, @Setter) et utilisation

```
import lombok.AllArgsConstructor; import lombok.NoArgsConstructor;
import lombok.Getter; import lombok.Setter; import lombok.ToString;
//----- lombok generation code annotations -----
@Getter @Setter
@ToString
//@EqualsAndHashCode
@NoArgsConstructor @AllArgsConstructor
//-----
@Entity @Table(name="Customer")
public class Customer {
    @Id
    private Long id;
    private String name;
}
```



# IV - Debug eclipse

## 1. Debug avec eclipse

Remarque préliminaire : on est pas obligé de programmer des bugs !!!

### 1.1. Mode opératoire classique pour le "debug"

1) Lancer le programme normalement (*Run as / java application*) et tenir compte des messages d'erreurs remontés (dans console ou fichier de log) :

```
Exception in thread "main" 85 6 1 32 7 95 3 java.lang.ArrayIndexOutOfBoundsException: 7
    at tp.TriABulle.triBulle(TriABulle.java:16)
    at tp.TriABulle.main(TriABulle.java:8)
```







2) Placer ensuite **un point d'arrêt sur la ligne de code à priori adéquate** (ici la ligne 16 de la méthode triBulle). Ceci s'effectue en *cliquant dans la marge* . Un point d'arrêt ("breakpoint") est alors matérialisé par un petit rond dans la marge (juste devant le numéro de ligne).

("toggle breakpoint" / "double click in margin" : place ou enlève un point d'arrêt)

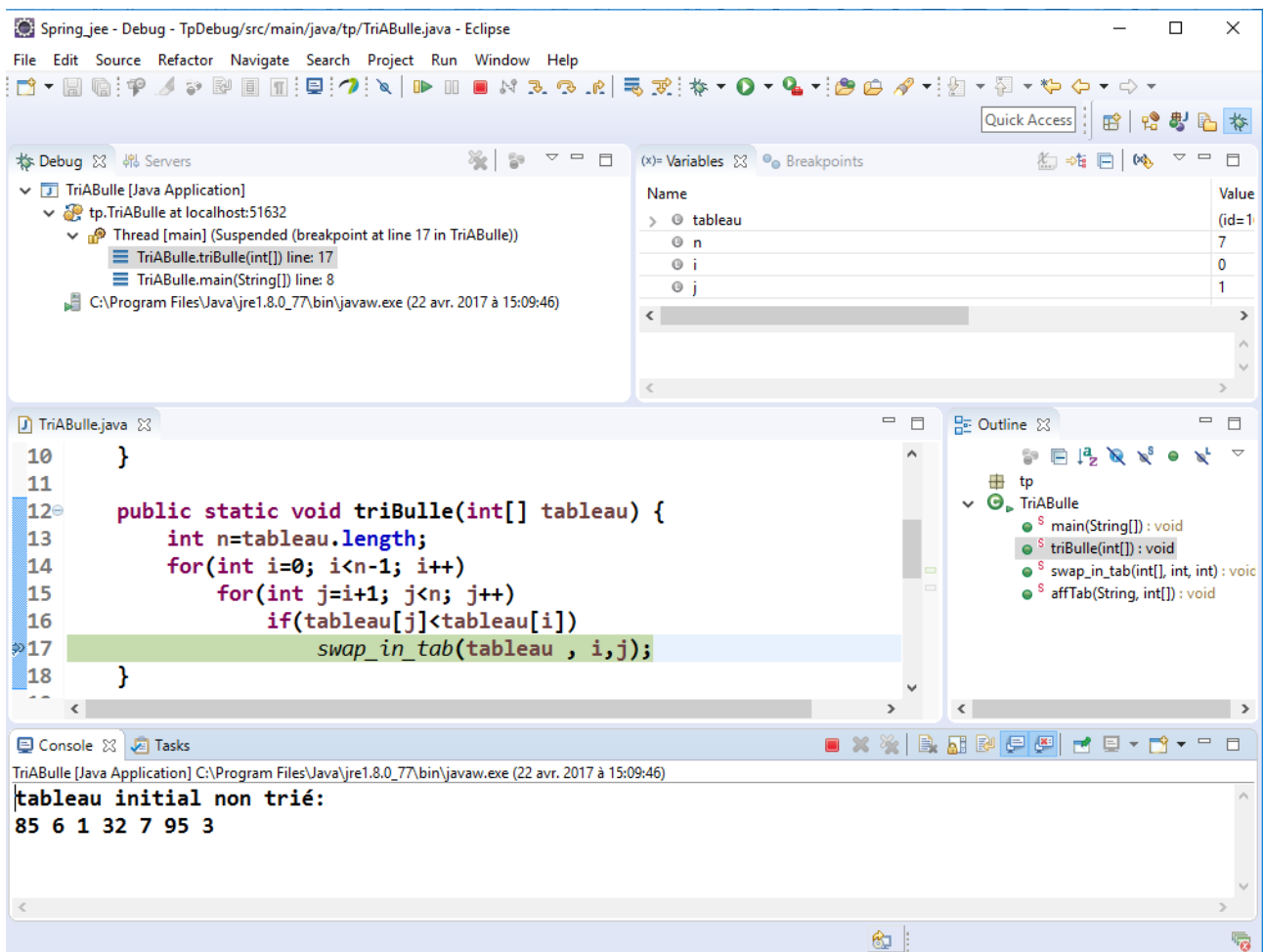
3) Relancer ensuite le programme en mode "debug" via le menu "*debug as / java application*" .

Eclipse bascule alors sur la **perspective "debug"** qui permet d'analyser "pas à pas" le comportement de l'application.

L'application s'exécute normalement jusqu'au point d'arrêt et l'on peut ensuite contrôler et surveiller comme on le souhaite l'exécution du code.

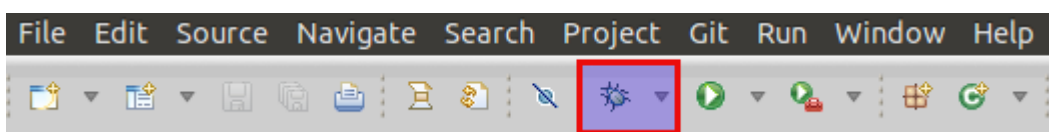
	<b>Resume (F8)</b>	Exécuter le code jusqu'au prochain point d'arrêt (ou la fin de l'exécution)
	Suspend	Effectuer une pause (suspendre l'exécution du code là ou il est)
	<b>Terminate (Ctrl+F2)</b>	Arrêter l'exécution du code (de l'application)
	Step into (F5)	Entrer dans l'exécution des lignes de code d'une méthode (si eclipse peut accéder au code source)
	<b>Step over (F6)</b>	Passer à l'instruction suivante (sans rentrer dans une sous fonction)
	Step return (F7)	Revenir à la méthode appelante
...		

NB : il est possible de demander au débogueur d'exécuter toutes les instructions jusqu' une ligne donnée (si tant est que cela soit possible) en se plaçant sur la ligne souhaitée et en tapant **CTRL-R** (qui est un raccourci pour **Run to Line** du menu *Run*).



- La vue "**debug**" permet de savoir quel thread exécute quelle partie du code à l'instant t et permet de ***naviguer dans la pile d'exécution*** : **en cliquant sur une des méthodes (qui s'appellent les unes les autres) , on affiche dans la vue "variables" , les variables locales de la méthode sélectionnée** .
- La vue "**Outline**" est une vue sur la structure du code (packages, classes, méthodes)
- La vue principale correspond aux lignes de code actuellement exécutées (souvent près d'un point d'arrêt)
- Les vues "**variables + breakpoints**" permettent de visualiser l'état des variables et de gérer finement la liste des point d'arrêts .

NB : En cliquant sur l'icône "debug" , on peut rapidement relancer un debug préalablement lancé (au moins une fois) via "debug as / java application" .



## 1.2. Affichage des variables "static"

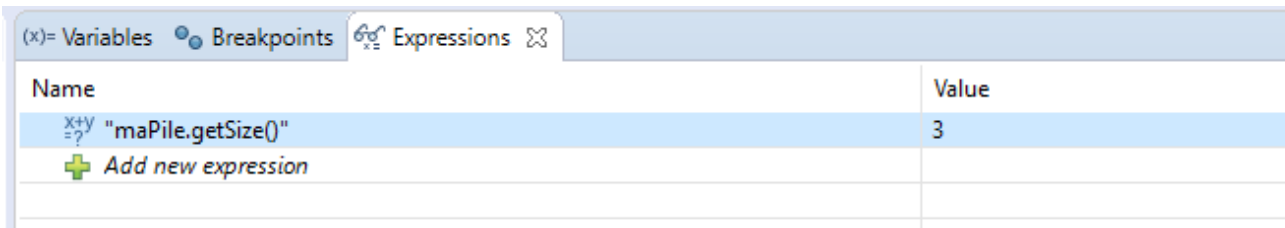
Par défaut, les variables affichées sont celles de la pile d'exécution d'un thread.

Dans la vue "**variables**", on peut cliquer sur le triangle orienté vers le bas puis déclencher le menu contextuel "**java / show static variables**".

Ceci permet évidemment d'afficher en plus les variables statiques.

## 1.3. Expressions (watch)

Via le menu habituel "**windows / show view /.../debug /expressions**" (ou *indirectement* via le menu contextuel "**watch**" depuis une partie d'une ligne de code), on peut faire apparaître une vue permettant de **surveiller** (et réactualiser en temps réel) **certaines expressions libres à saisir** (exemple : `objXy.getName()`) de façon à se focaliser sur quelques valeurs pertinentes :

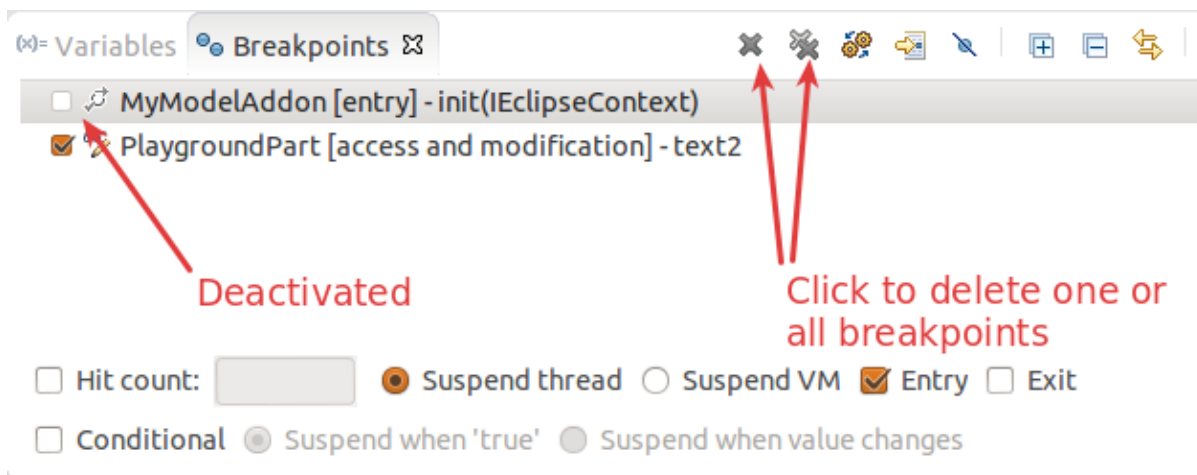


## 1.4. Gestion fine des points d'arrêts

Via un click droit sur un point d'arrêt (rond dans la marge) et le menu contextuel "**breakpoint properties**", il est possible de paramétrer finement un point d'arrêt (ce qui n'est utile que dans les cas pointus) :

- **Enabled/disabled** : en mode "disabled", un point d'arrêt est ignoré (mais pas supprimé) : on peut rapidement le réactiver par la suite.
- **Hit Count** : permet d'indiquer le nombre de fois où l'on peut passer sur le point d'arrêt sans que le programme (thread) soit suspendu (pratique en cas de boucle).
- **Enable Condition** : permet d'indiquer une condition sous la forme d'une expression booléenne, ou pour un changement de valeur d'une variable (lorsqu'un point d'arrêt est conditionné, un point d'interrogation est accolé son icône).

NB : Les points d'arrêts peuvent également être gérés dans l'onglet (vue) "Breakpoints" :



## 1.5. WatchPoint (breakpoint sur attribut/field)

En se plaçant sur la partie attributs d'une classe java et en double cliquant dans la marge (devant l'attribut qui nous intéresse), on peut ajouter un "watchPoint" (paramétrable en "*Field Access*", "*Field Modification*" ou les deux) qui sera prise en compte de la façon suivante :

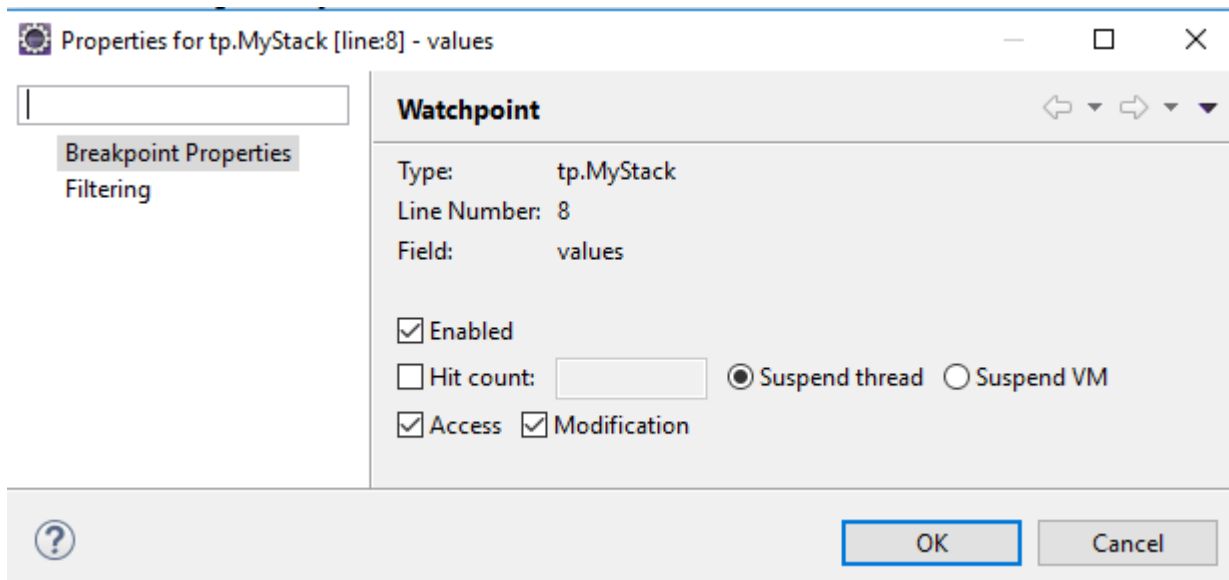
Lors de l'exécution du code , le débogueur sera suspendu (et près à être contrôlé) dès que la valeur de l'attribut associé au "watchPoint" sera soit lue ou modifiée (selon paramétrage) .

```

6 public class MyStack {
7
8     private List<String> values = new ArrayList<String>();
9
10    public void push(String val){
11        values.add(val);
12    }
13
14    public String pop(){
15        String res = null;
16        int s = values.size();
17        if(s>0) res = values.remove(s-1);
18        return res;
19    }

```

*./breakPoint properties (pour watchPoint) :*



## 1.6. Autres types de "breakPoints" :

<i>Types de "breakpoint"</i>	<i>Paramétrages</i>	<i>Comportements</i>
<b>Exception breakpoint</b>	<b> Icône (!)</b> "add java breakpoint exception" dans barre d'outils de la vue "breakpoints" .	Thread d'exécution suspendu dès la levée d'une exception
<b>Method breakpoint</b>	Double click devant signature d'une méthode. Method <b>Entry</b> , Method <b>Exit</b>	Thread d'exécution suspendu dès l'entrée et/ou la sortie d'une méthode
<b>Class Loading breakpoint</b>	Double click devant une classe dans la vue "outline"	Suspension au chargement d'une classe en mémoire .

NB : via le menu "*window / preferences / java / debug / Step filtering*" , on peut préciser que certains packages (et tous leurs contenus) seront ignorés en cas de "F5" (step into) .

## 2. Debug web (remote debug)

### 2.1. Debug web en mode manuel / externe

1) Déployer le ".war" de l'application dans le répertoire "webapps" de tomcat et placer au moins un point d'arrêt dans une page JSP ou bien un servlet

De façon à indirectement démarrer (au dehors d'eclipse) tomcat avec l'option suivante

`-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000`

il suffit d'écrire et lancer le script suivant dans le répertoire bin de tomcat :

***startTomcatInDebugMode.bat***

**catalina jpda start**

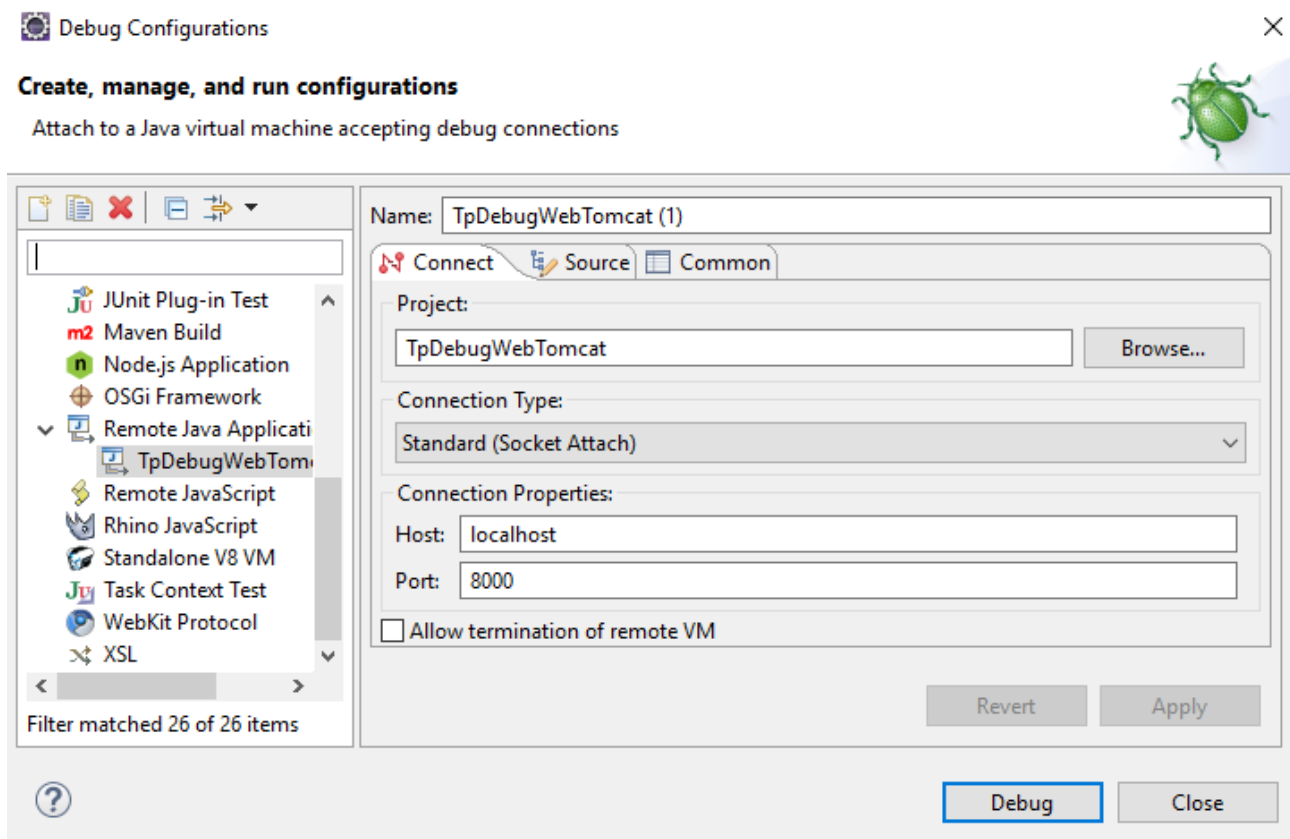
→ tomcat démarre alors en mode "debug" avec l'application déjà installée dans webapps

**Configurer le "remote debug" eclipse de la façon suivante :**

**Run > Debug Configuration**

**Remote Java Application**

**New** (appName + port=8000)



il faut ensuite cliquer sur l'icône "debug" (dans run configuration)

via un navigateur web ordinaire (firefox, chrome, ...) utiliser l'application normalement (exemple : <http://localhost:8080/TpDebugWebTomcat>)

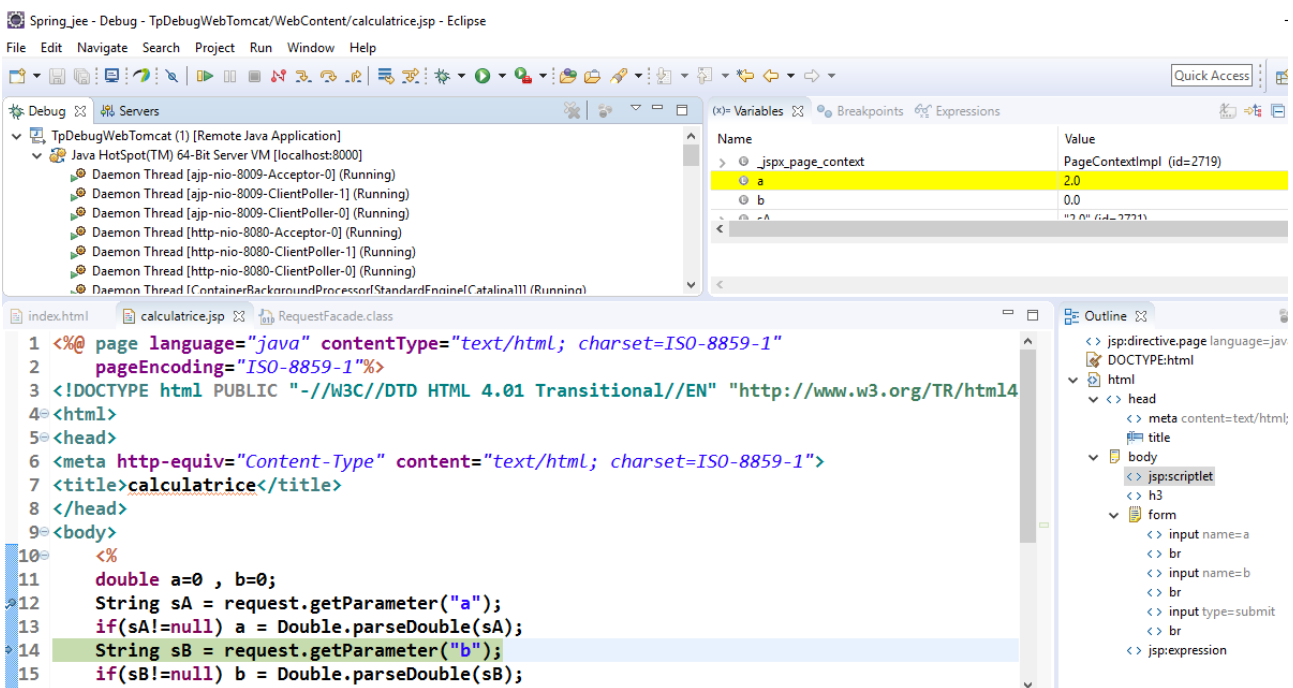
### calculatrice

a:

b:

a+b: 8.0

→ Un thread de tomcat sera alors automatiquement suspendu lorsque le point d'arrêt sera atteint et l'on aura le contrôle habituel dans la perspective "debug" de eclipse ( "Step over" , .... )



## 2.2. Debug web en mode automatique (intégré à eclipse)

- 1) placer un point d'arrêt dans une page JSP ou bien un servlet
- 2) lancer l'application via **debug as / run on server**

Tout simplement !

# V - Maven et eclipse

## 1. Présentation de maven

**Apache Maven** est un outil logiciel open source pour la gestion et l'automatisation de production des projets logiciels Java/JEE .

**Maven** utilise un paradigme connu sous le nom de **Project Object Model (POM)** afin de décrire un projet logiciel, ses dépendances avec des modules externes. Il est livré avec un grand nombre de tâches pré-définies, comme la compilation de code Java .

Chaque projet ou sous-projet est configuré par un fichier **pom.xml** à la racine du projet qui contient les informations nécessaires à Maven pour traiter le projet ( nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, url des référentiels, ...).

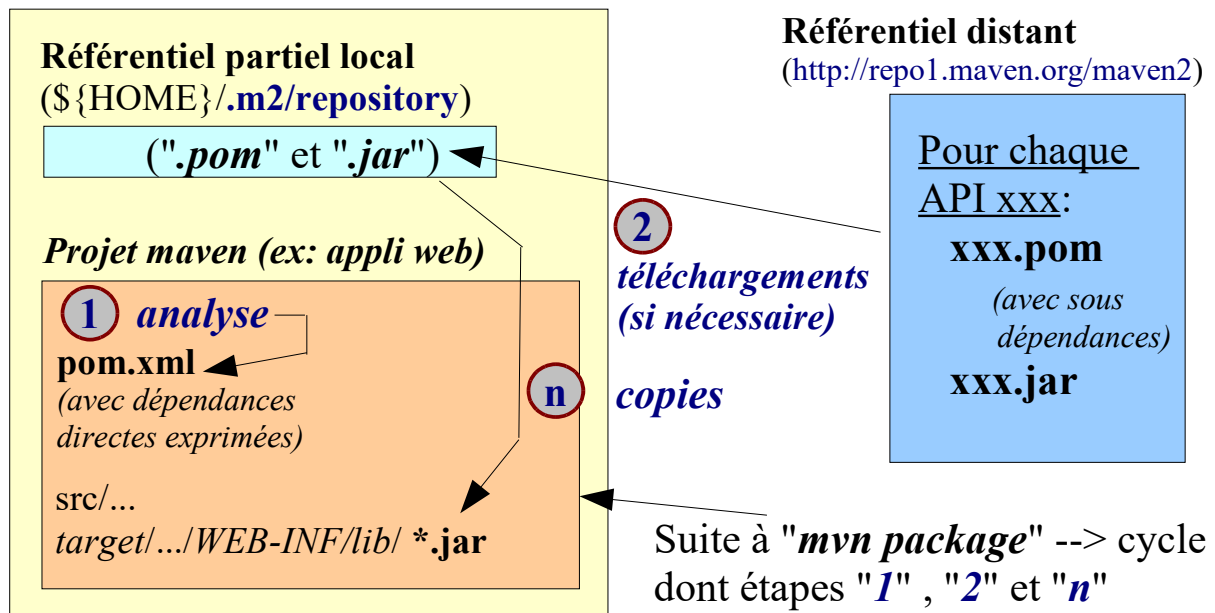
Maven impose une arborescence et un nommage des fichiers du projet selon le concept de **Convention plutôt que configuration**.

### Maven (principales fonctionnalités)

- **Centré sur la notion de projet** (api java, module applicatif)  
--> toutes les caractéristiques d'un projet sont déclarées dans un fichier "**pom.xml**" (**P**roject **O**bject **M**odel).
- **Configuration "déclarative"** (basée sur des conventions) plutôt qu'explicite. Pas de commandes et chemins précis à renseigner (contrairement à ANT).
- **Gestion distribuée** (sur le web) des **dépendances** inter-projets.  
---> identification et **téléchargement automatique des ".jar"** nécessaires selon les API déclarées en dépendances .
- Gère toutes les phases (compile/build , tests unitaires , packaging , stockage dans le référentiel , éventuel lien avec SVN, ...) .



## Téléchargement automatique des librairies nécessaires (avec prise en compte des dépendances indirectes)



### 1.1. Versions des artifacts "maven"

Par convention, une pré-version en cours de développement d'un projet voit son numéro de version suivi d'un -SNAPSHOT.

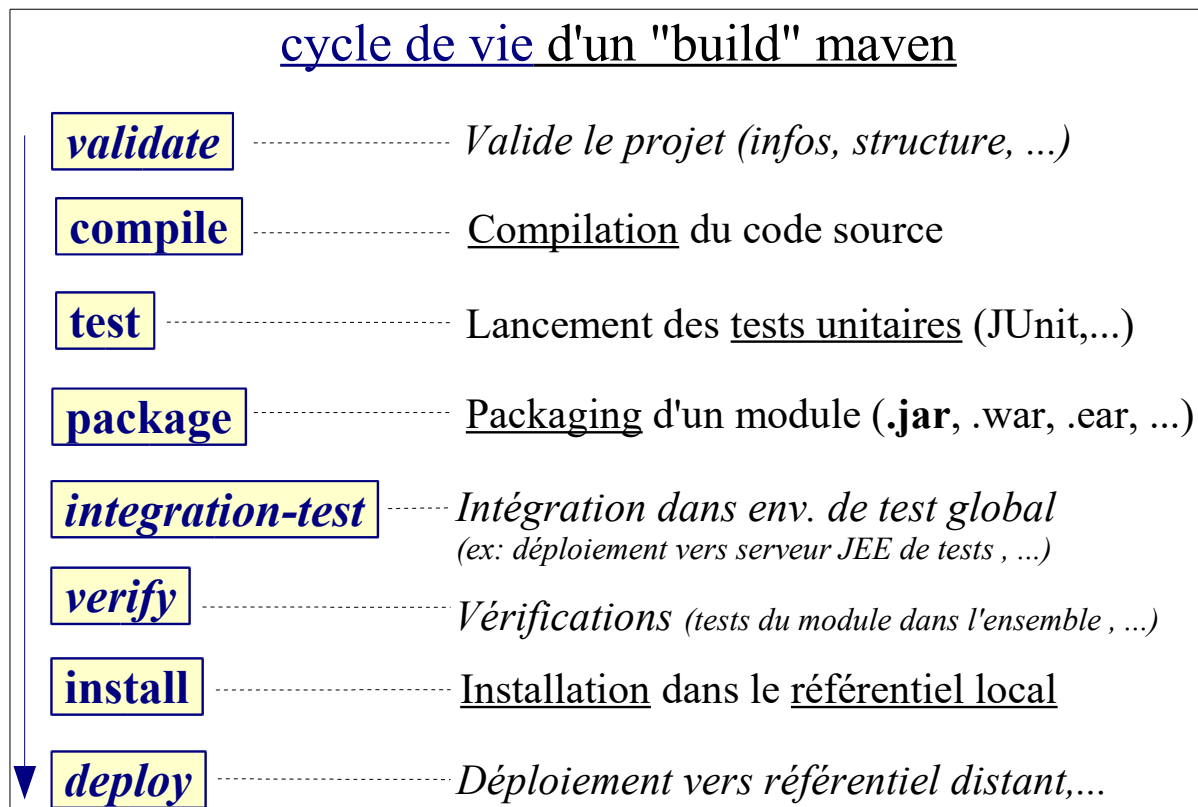
Dans la gestion des dépendances, Maven va chercher à mettre à jour les versions SNAPSHOT régulièrement pour prendre en compte les derniers développements.

Utiliser une version SNAPSHOT permet de bénéficier des dernières fonctionnalités d'un projet, mais en contre-partie, cette version peut être appelée à être modifiée de façon importante, sans aucun préavis.

## 2. Cycle de build , tâches et plugins

### 2.1. "buts/goals" liés au cycle de construction d'un projet maven

Principaux buts (goals):



### Phases du cycle de construction

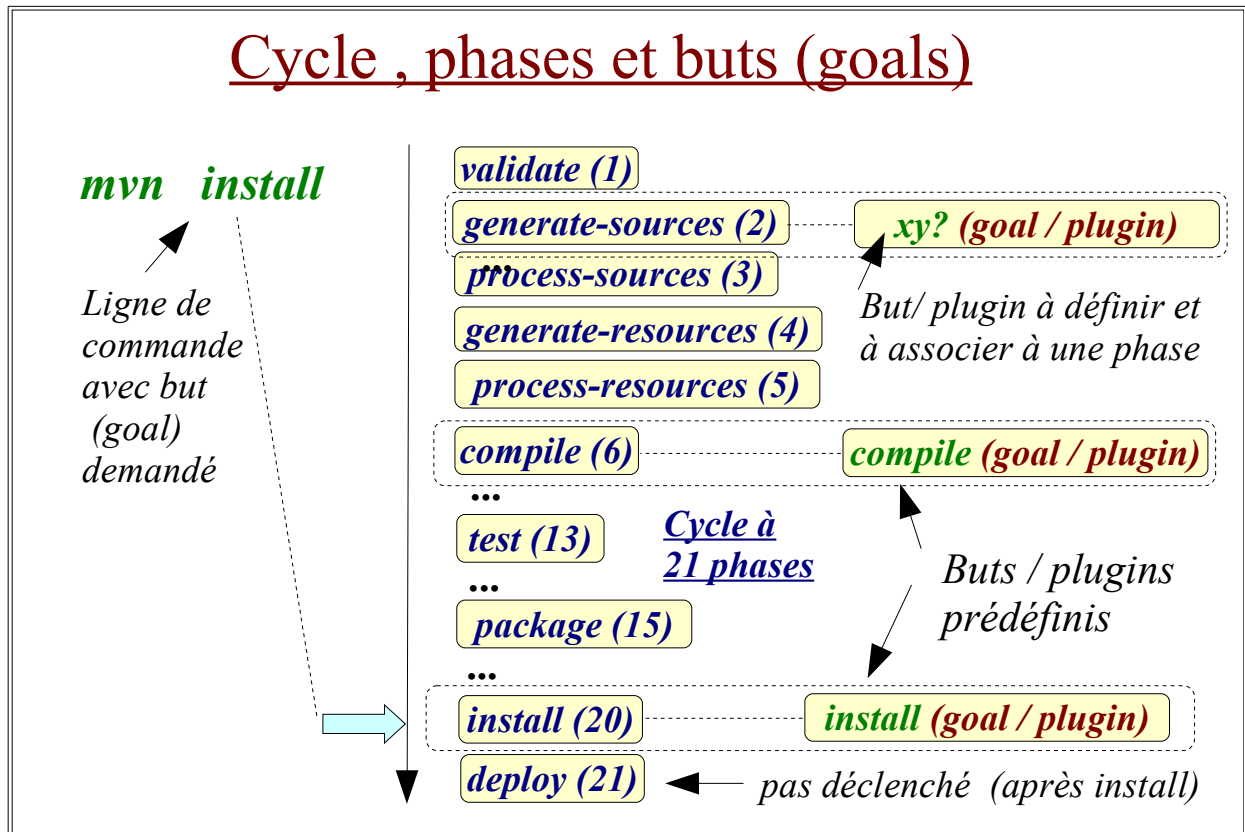
Lorsque l'on déclenche une *ligne de commande* "**mvn <goal>**" (ex: **mvn install**) , on *demande explicitement à atteindre un but* .

Pour atteindre ce but , maven va déclencher un processus de construction qui comporte au maximum 21 phases (dans la version actuelle).

Ces phases ont des "ordres" et "noms" bien déterminés (ex: **validate(1)** , **generate-sources (2)** , ...).

Selon le paramétrage du projet (pom.xml) , les phases de 1 à n-1 seront (ou pas) associées à des plugins (actions / buts préalables) à déclencher.

La demande d'atteinte du but associé à la phase n , déclenche dans l'ordre l'exécution de tous les plugins associés aux phases 1 , ..., n-1 puis n .

lien entre phases et buts (goals):Phases du cycle de construction par défaut:

Plugins ( et goals) prédéfinis et activés lors du cycle par défaut (pour packaging "jar").

Nom de la phase	plugin:goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar (*)
install	install:install
deploy	deploy:deploy

(\*) ou war:war , ejb:ejb3 , ear:ear ... selon autre type de packaging du projet .

NB:

- Chaque but (goal) est codé dans un plugin maven (packagé comme un ".jar") .
- Un plugin maven peut contenir plusieurs buts (goals)  
[ ex: deploy:deploy , deploy:deploy-file , deploy:....]

### 3. Structure d'un projet et conventions

#### 3.1. GroupId & artifactId

##### groupId & artifactId (quelques exemples)

*Éditeur,  
organisation/entreprise  
(+éventuelle sous branche)*

*Produit (application, sous module, Api)*

org.apache.cxf ————— cxf-api , cxf-rt-core , ...

org.springframework ——— spring-core , spring-orm , ...

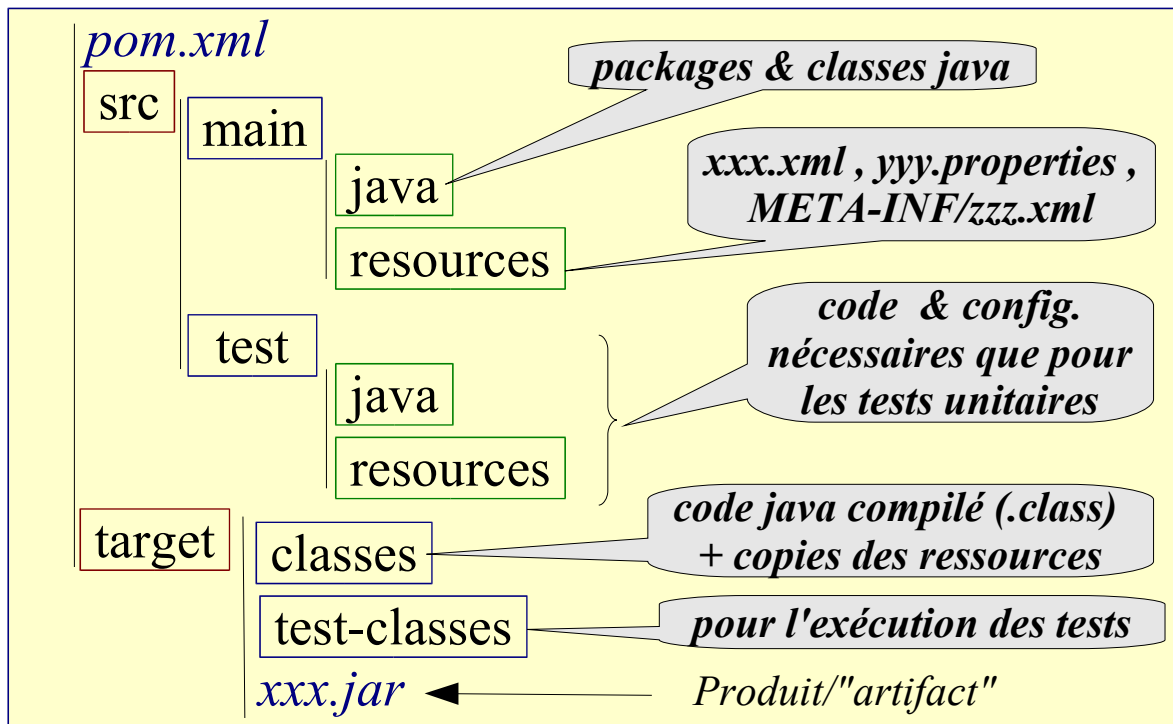
org.hibernate ————— hibernate-core, hib...-annotations, ...

javax.persistence ——— persistence-api (JPA)

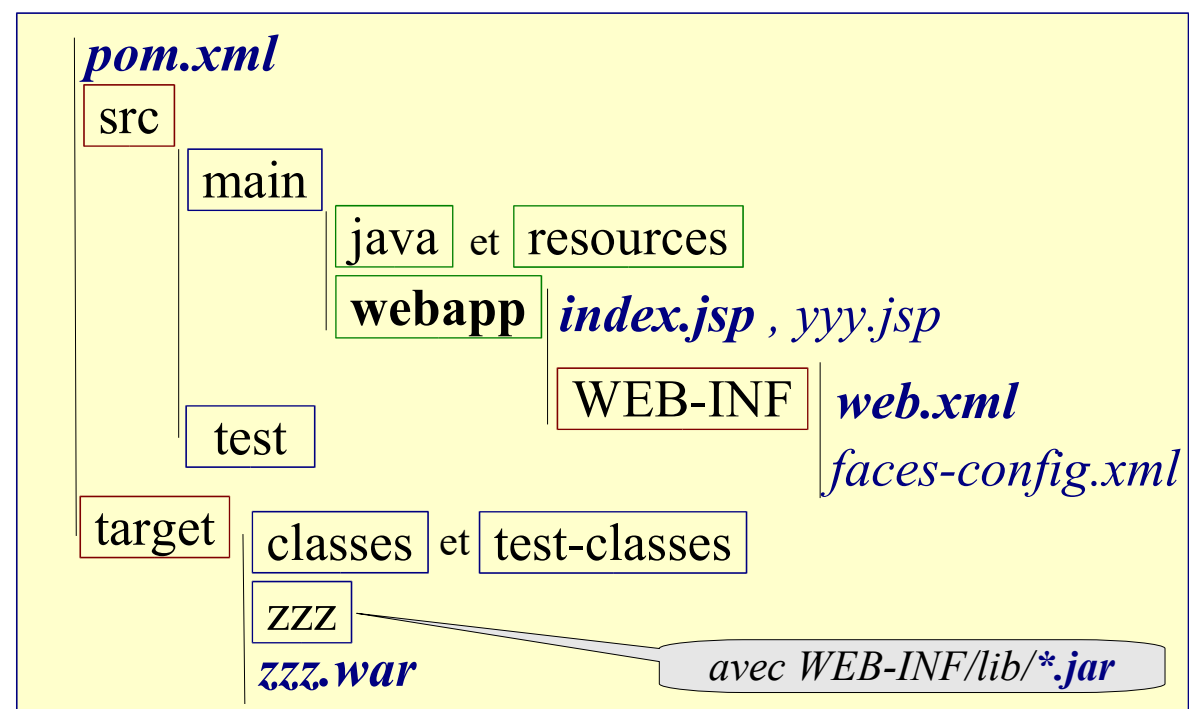
javax.servlet ————— servlet-api

### 3.2. Arborescences conventionnelles

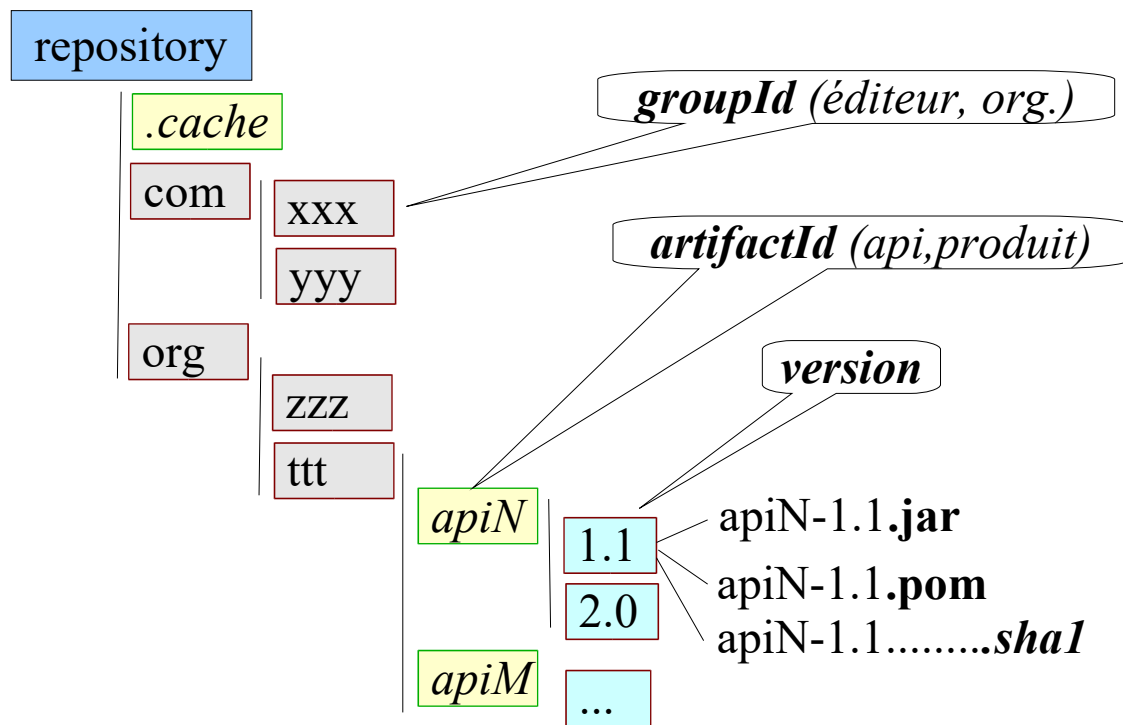
#### Structure (quasi-imposée) par conventions "maven"



#### Structure supplémentaire pour module "java/web"



## Structure d'un référentiel "maven" (local ".m2" ou distant)



### 3.3. portées ("scope") des dépendances:

#### Principaux types de dépendances "maven" (*scope*)

- . **compile** *(par défaut)*  
 --> **nécessaire pour l'exécution et la compilation** (dépendance directe puis transitive) [diffusé dans tous les "classpath"].
- . **runtime**  
 --> **nécessaire à l'exécution** (dépendance indirecte **transitive**)
- . **provided**  
 --> **nécessaire à la compilation** mais **fourni par l'environnement d'exécution (JVM + Serveur JEE)** [diffusé uniquement dans les "classpath" de compilation et de test, dépendance non transitive]
- . **test**  
 --> uniquement nécessaire pour les **tests** (ex: *spring-test.jar* , *junit4.jar*)

### Diffusé dans quel(s) "classpath" ?

Type de dépendances	compilation	Tests unitaires	exécution	Transitivité (dans futur projet utilisateur / propagation)
compile (C)	x	x	x	C(C) -->C(*), P(C) -->P T(C) -->T, R(C) -->R
provided (P)	x	x	x (provided)	--> pas propagé , à ré-expliciter si besoin
test (T)	x	x		--> pas propagé
runtime (R)		x	x	R(R) --> R, C(R)-->R T(R)-->T, P(R)-->P

(\*) bizarrement quelquefois "compile" plutôt que "runtime" dans le cas où l'on souhaite ultérieurement étendre une classe par héritage.

## 3.4. Structure & syntaxes (pom.xml)

### Fichier "POM" (éléments essentiels)

*servlet-api-2.3.pom (exemple)*

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.3</version>
</project>
```

← Version du modèle interne de maven

*biblio-web-....pom*

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
  <groupId>tp</groupId> <artifactId>biblio-web</artifactId>
  <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
  <name>biblio-web JEE5 Webapp</name>
  .... <dependencies> ... </dependencies> <build>...</build>
</project>
```

## Fichier "POM" (déclaration des dépendances / partie 1)

```

<project ...>
  <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
  <groupId>tp</groupId> <artifactId>biblio-web</artifactId>
  <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
  <name>biblio-web JEE5 Webapp</name>....
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.5.6</version>    <scope>compile</scope>
    </dependency> ...
  </dependencies>
  <build>....</build>
</project>

```

## Fichier "POM" (déclaration des dépendances / partie 2)

```

...
<dependency>
  <groupId>org.hibernate</groupId> <artifactId>hibernate-core</artifactId>
  <version>3.5.1-Final</version> <scope>compile</scope>
  <exclusions>
    <exclusion>
      <groupId>javax.transaction</groupId>
      <artifactId>jta</artifactId>
    </exclusion>
    <exclusion>
      <groupId>asm</groupId> <artifactId>asm</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>javax.transaction</groupId>
  <artifactId>jta</artifactId> <version>1.1</version>
</dependency>
...

```

exclusion(s)  
explicite(s) de  
dépendance(s)  
indirecte(s)  
transitive(s)

Contrôle direct  
de la version  
souhaitée pour  
éviter des conflits  
ou des doublons



## Mise au point des dépendances (partie 3)

- La ***mise au point des dépendances*** peut éventuellement être délicate en fonction des différents points suivants:
  - \* potentiel ***doublon*** (2 versions différentes d'une même librairie) à partir de plusieurs dépendances transitives indirectes.
  - \* potentiel ***conflit*** de librairie à l'exécution (incompatibilité entre une librairie "A" en version "runtime" et une librairie complémentaire "B" en version "provided" imposée par le serveur JEE)
  - \* autres mauvaises surprises de "murphy" .
- Eléments de solutions:
  - \* ***étudier finement les compatibilités/incompatibilités*** et re-paramétrer les "***version***" et "***exclusion***"
  - \* (tester , ré-essayer , re-tester) de façon itérative

## Fichier "POM" (partie "build")

```

<project ...>
  <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
  <groupId>tp</groupId> <artifactId>biblio-web</artifactId>
  <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
  <name>biblio-web JEE5 Webapp</name>....
  <dependencies>
    <dependency>...</dependency> ...
  </dependencies>
  <build>
    <plugins> <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId><version>2.0.2</version>
      <configuration>
        <source>1.6</source> <target>1.6</target>
      </configuration>
    </plugin>... </plugins>
    <finalName>biblio-web</finalName>
  </build>
</project>

```

## Fichier "POM" (parties "repositories" et "properties")

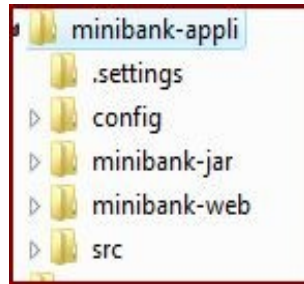
```

<project ...>
  <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
  <groupId>tp</groupId> <artifactId>biblio-web</artifactId>
  <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
  <name>biblio-web JEE5 Webapp</name>....
  <repositories> <!-- en plus de http://repo1.maven.org/maven2 -->
    <repository> <!-- specific repository needed for richfaces -->
      <id>jboss.org</id>
      <url>http://repository.jboss.org/maven2/</url>
    </repository> ...
  </repositories>
  <properties>
    <org.springframework.version>3.0.5.RELEASE</org.springframework.version>
    <org.apache.myfaces.version>2.0.3</org.apache.myfaces.version>
  </properties>
  <dependencies> <dependency>...
    <version>${org.springframework.version}</version>
  </dependency> ...</dependencies> <build>....</build>
</project>

```

### 3.5. Configuration multi-modules (avec sous projet(s))

Mode "multi-projets" [ parent/enfants , ear(war,jar) ]



#### pom.xml (mod. web)

```
<project ....>...
  <parent> ...
    <artifactId>minibank-appli</artifactId>
  </parent> <groupId>tp</groupId>
  <artifactId>minibank-web</artifactId>
  <packaging>war</packaging>
  <dependencies>
    <dependency>
      <groupId>tp</groupId> ...
      <artifactId>minibank-jar</artifactId>
      <scope>runtime ou compile</scope>
    </dependency> ...<dependencies>...
  </project>
```

#### pom.xml (parent)

```
<project ....>...
  <artifactId>minibank-appli</artifactId>
  <packaging>pom</packaging>
  <modules>
    <module>minibank-jar</module>
    <module>minibank-web</module>
  </modules>
</project>
```

#### pom.xml (sous module de services)

```
<project ....>...
  <parent>
    <artifactId>minibank-appli</artifactId>
    <groupId>tp</groupId> ...
  </parent>
  <groupId>tp</groupId>
  <artifactId>minibank-jar</artifactId>
</project>
```

## Mode "multi-modules" (suite)

Arborescence globale conseillée:

### **my-global-app**

pom.xml

**minibank-jar** (module de services)

...

pom.xml

**mywebapp**

pom.xml

...

*Un module retrouve son projet parent dans le répertoire parent « .. »  
tandis qu'un projet retrouve son éventuel projet parent dans un référentiel maven (local ou distant).*

Principal intérêt du mode multi-module :

- \* **ne pas devoir construire de multiples projets séparés un par un (dans l'ordre attendu en fonction des dépendances)**
- \* **simplement lancer la construction du projet principal pour que tous les sous-modules soient automatiquement (re-)construits dans le bon ordre (selon inter-dépendances)**

**pom.xml** (de niveau projet global)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-global-app</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>...</name>
  <modules>
    <module>services</module> <!-- ou <module>../services</module> -->
    <module>mywebapp</module> <!-- ou <module>../mywebapp</module> -->
  </modules>
</project>
```

....

**pom.xml** (de niveau sous projet / module)

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>mywebapp</artifactId> <!-- même nom que sous module courant -->
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging> <!-- ou jar -->
  ...
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-global-app</artifactId>
  </parent>
  ...
  <dependencies>
    <!-- ici le module de présentation (ihm web) utilise le module frère "services"
    et la dépendance sera alors interprétée comme une dépendance directe (source)-->
    <dependency>
      <groupId>${pom.groupId}</groupId>
      <artifactId>services</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    ...
  </dependencies>
</project>

```

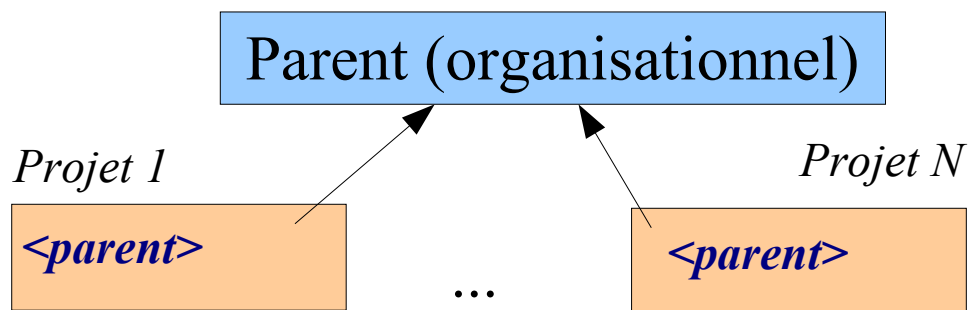
### 3.6. Héritage entre projets "maven" (<parent>)

#### Héritage "organisationnel" au niveau de "maven"

Dans l'absence d'une organisation multi-modules, la balise **<parent>** d'un fichier "**pom.xml**" permet de définir un lien d'héritage entre le projet courant et le projet parent :

*Une certaine partie de la configuration du projet parent est ainsi héritée (sans devoir être répétée).*

--> intérêt du projet parent: **factoriser** une **configuration commune** entre différents projets "fils".

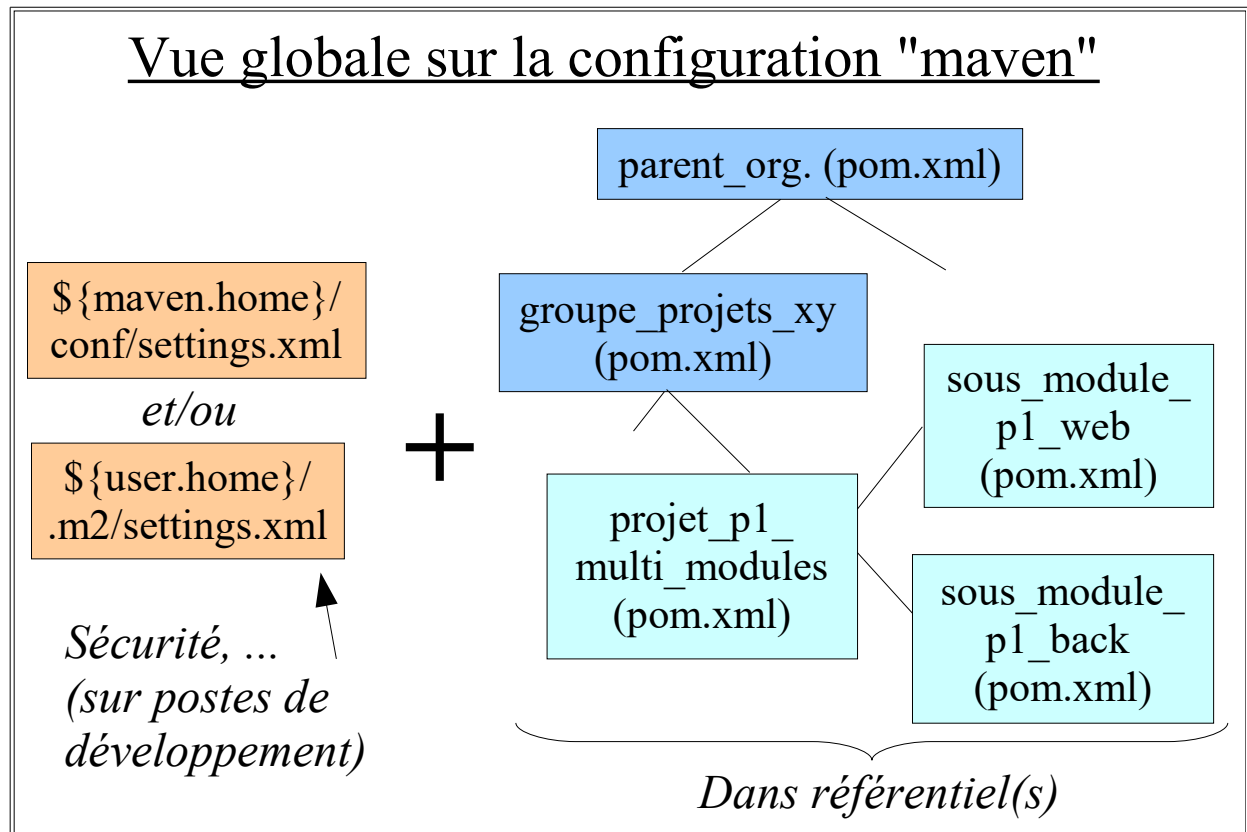


#### Éléments couramment hérités (pom.xml)

Un projet parent "organisationnel" (*sans obligation d'être placé dans un répertoire parent*) sert généralement à factoriser les points suivants:

- ◆ *des informations générales sur l'entreprise (organisation, e-mails).*
- ◆ *liste ( avec URL) des référentiels "maven" et "svn/git" de l'entreprise et/ou des référentiels externes (<repositories> , <distributionManagement> , ...) .*
- ◆ *section <dependencyManagement> servant à préciser des versions et des exclusions au sein des futures dépendances qui seront exprimées au cas par cas dans les projets fils.*
- ◆ ....

Conseil: plusieurs niveaux de parents (parent intermédiaire).

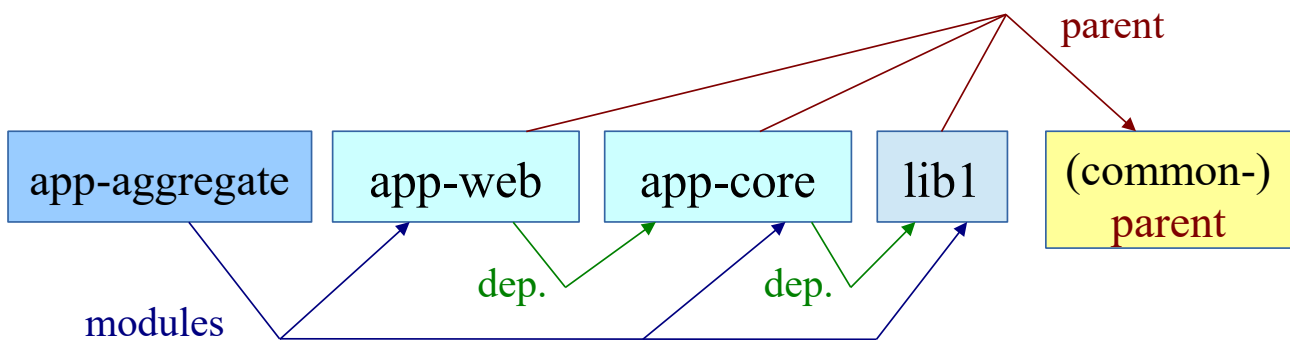


### Notion de projet "**aggregate**" (depuis maven 3)

Un projet "**aggregate**" de type "pom" *sert simplement à tout reconstruire d'un coup* via "mvn install" .

Les mécanismes "reactor" de maven vont **tenir compte des inter-dépendances** entre les projets référencés comme des modules et **tout construire DANS LE BON ORDRE**.

Cependant, contrairement à un projet multi-modules classique **les projets référencés ne sont pas des sous-répertoires** mais **des répertoires de même niveau** et ces différents projets ont un **autre parent** que le projet courant "app-aggregate"



**app-aggregate/pom.xml**

```

<project ...>... <packaging>pom</packaging>
  <artifactId>app-aggregate</artifactId> <version>0.0.1-SNAPSHOT</version>
  <modules>
    <module>../common-parent</module>
    <module>../lib1</module>
    <module>../app-core</module>
    <module>../app-web</module>
  </modules> </project>

```

**app-core/pom.xml**

```

... <parent>...
<artifactId>common-parent</artifactId>
<version>0.0.2-SNAPSHOT</version>
<relativePath>../common-parent
</relativePath>
</parent>
<groupId>tp</groupId>
<version>0.0.1-SNAPSHOT</version>
<artifactId>app-core</artifactId>

... <dependency>
  <groupId>tp</groupId>
  <artifactId>lib1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency> ...

```

**app-web/pom.xml**

```

<parent> ...
<artifactId>common-parent</artifactId>
<version>0.0.2-SNAPSHOT</version>
<relativePath>../common-parent
</relativePath>
</parent>
<groupId>tp</groupId>
<version>0.0.1-SNAPSHOT</version>
<artifactId>app-web</artifactId>
<packaging>war</packaging>
... <dependency>
  <groupId>tp</groupId>
  <artifactId>app-core</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency> ...

```



## 4. Lien entre maven et eclipse (m2e)

Etant donné que l'IDE eclipse gère lui aussi les projets "Java/JEE" avec sa propre structure de répertoires (différente de Maven) , **il faut installer au sein d'eclipse un plugin "Maven" spécifique de façon à ce qu'eclipse puisse déléguer à Maven certaines tâches** (compilations , gestion des dépendances , ....) .

*Attention:* Ce plugin s'appelle m2e ("maven2eclipse") , il n'est vraiment au point que dans ses versions les plus récentes (pour eclipse 3.5 et 3.6 , 3.7 , >=4.2) .

### Plugin eclipse "m2e" pour maven (partie 1)

Depuis eclipse , on peut installer le plugin "m2e" (maven to eclipse) via l'update site suivant: <http://m2eclipse.sonatype.org/sites/m2e> .

On peut ensuite créer de nouveaux projets eclipse de type "maven" via le menu habituel. Le choix d'un archetype peut être effectué dès la création du projet (mais n'est pas obligatoire).

Lorsque l'on restructure en profondeur le fichier pom.xml , il est conseillé de déclencher le menu contextuel "**Maven/Update Project Configuration**" d'eclipse pour que la configuration du projet eclipse s'adapte à la configuration "maven".

Le menu contextuel "**Run as ...**" d'eclipse permet de déclencher les "build" ordinaires de maven (*clean , test, package , install*, ...).

*NB : URL exacte à ajuster selon version d'eclipse*

### 4.1. Utilisation du plugin eclipse "m2e"

Installer si nécessaire dans eclipse 3.x ou 4.2 le plugin eclipse "m2e" via le "update site" suivant:

- <http://...../m2e/...>
- <http://...../m2e-wtp/...> [ *NB: m2e-wtp permet le "Run as / run on server" classique depuis un projet web sans trop d'erreur* ]

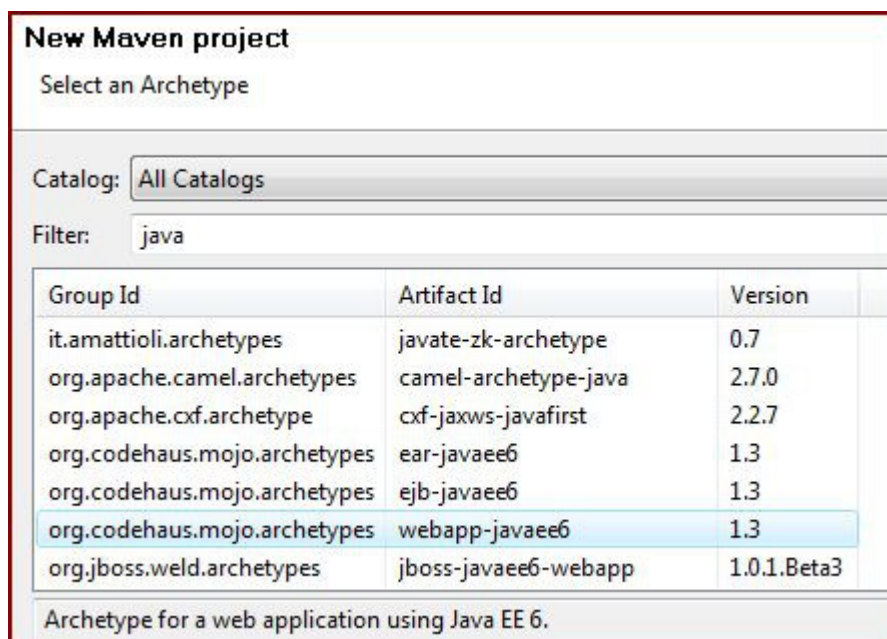
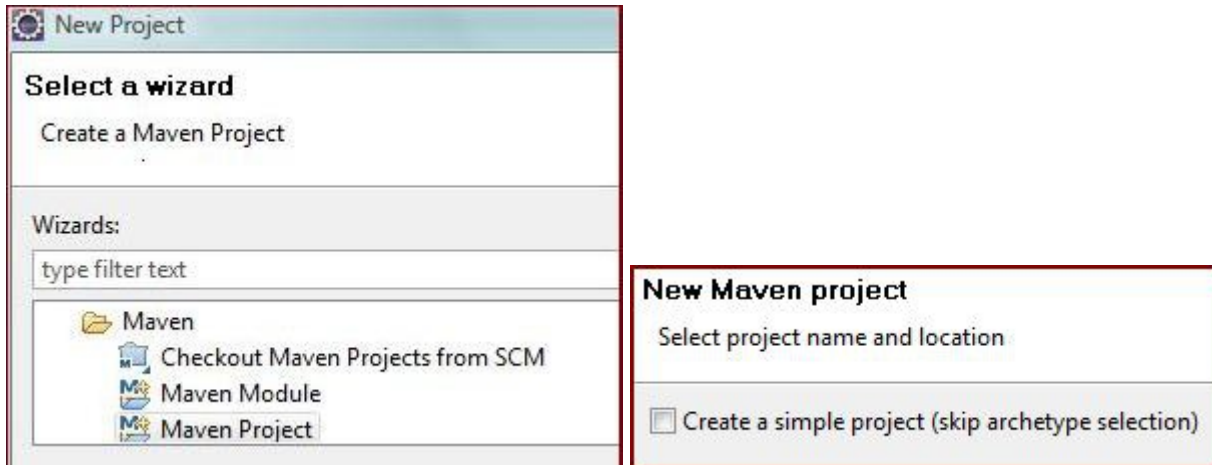
+ nouveau projet "maven"

+ éditer manuellement le fichier "pom.xml"

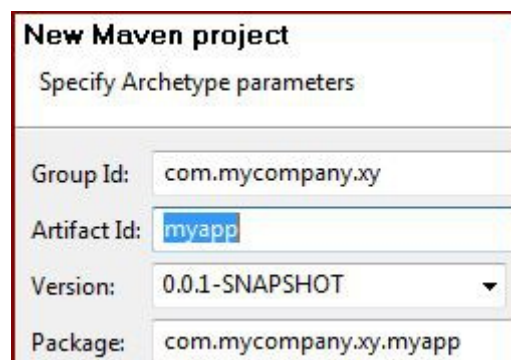
+ éditer le code au bon endroit (dans src/main/java, ....)

- **run as / maven .... sur le projet** (ou **run as / ...** sur des sous parties (test junit))

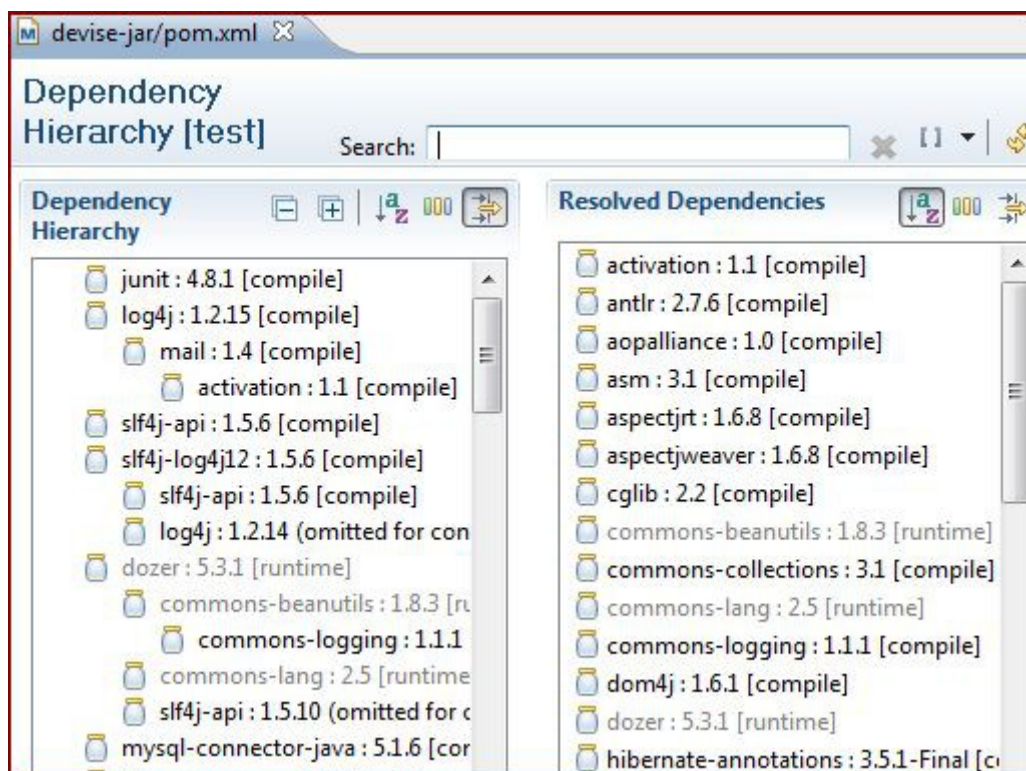
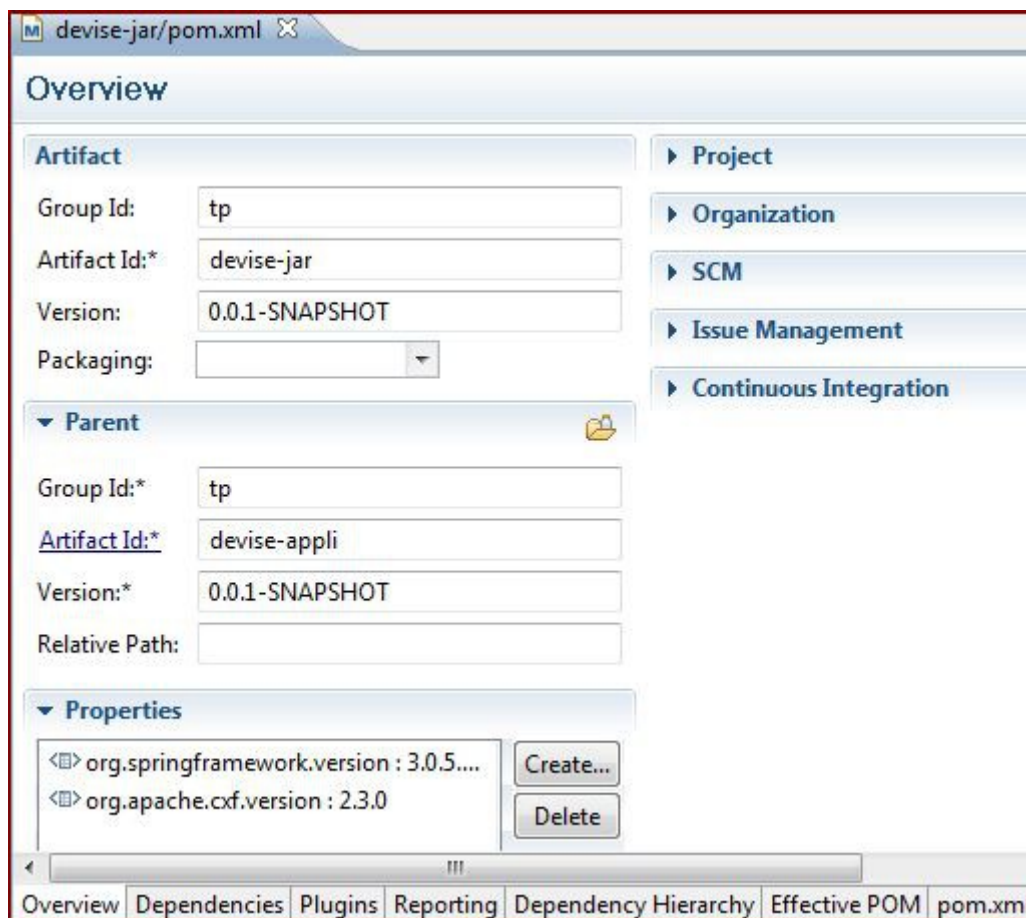
## 4.2. Assistants "eclipse/m2e" pour créer un nouveau projet maven



**NB :** Un archetype coorespond à un modèle de nouveau projet. Cela doit se préparer. **Tant qu'aucun archetype existant ne soit suffisamment intéressant, il est préférable de sauter la sélection d'un archetype en cochant la case "create simple project (skip archetype selection)".**



### 4.3. Assistants "eclipse/m2e" pour paramétrer/visualiser pom.xml



NB :

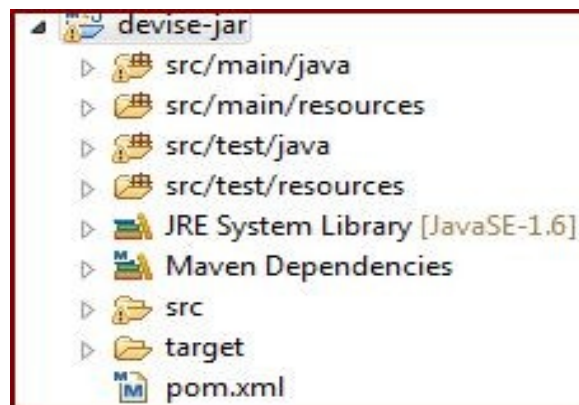
- l'onglet "**pom.xml**" correspond au contenu exact du fichier "pom.xml" du projet courant.
- l'onglet "**Effective POM**" correspond à **la somme du contenu de pom.xml et de toute la configuration héritée** (projet parent + config par défaut) .
- l'onglet "**Dependency Hierarchy**" permet de bien visualiser les dépendances indirectes ce qui permet quelquefois de choisir la meilleur version en cas de conflit de versions et/ou d'alléger la configuration en n'explicitant que les dépendances essentielles .

#### 4.4. Structure des projets "eclipse maven"

##### Plugin eclipse "m2e" pour maven (partie 2)

Via le plugin "m2e" , eclipse peut intégrer "maven" à travers des **projets "maven\_dans\_eclipse"** qui :

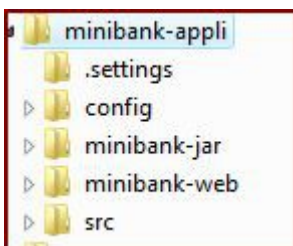
- \* ont une structure "maven" classique (*src/main/java, ...*)
- \* n'ont pas la structure classique d'un projet java (*src, bin*)  
ni la structure d'un "dynamic web projet" (*webContent,...*)



##### Cas d'une structure multi-modules:



(structure vue à plat dans eclipse)



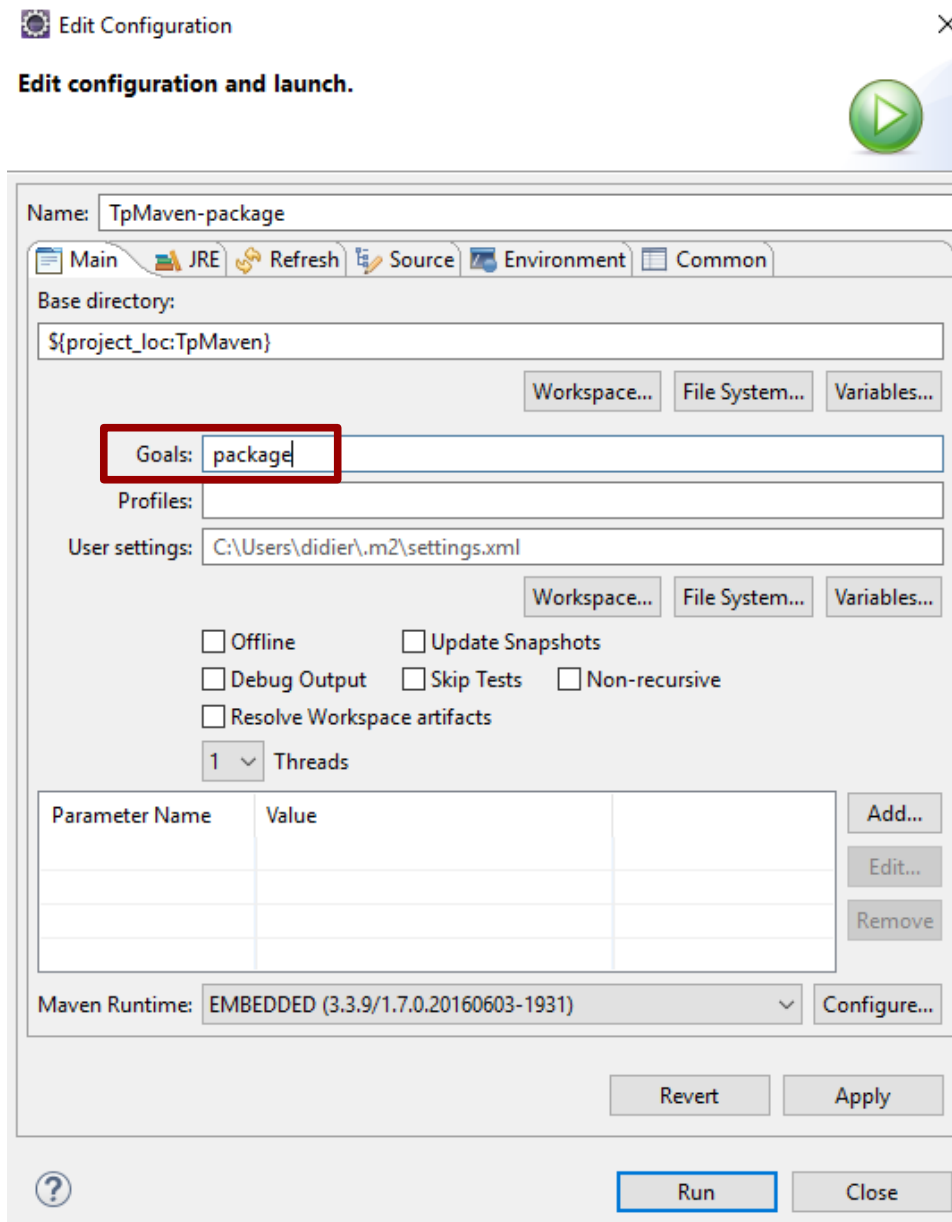
(structure arborescente sur le file system hôte).

## 4.5. Lancement d'un build maven depuis eclipse

Au sein des menus "**Run as / maven ....**" certains buts (classiques) sont directement accessibles :

- run as / **maven clean**
- run as / **maven compile**
- run as / **maven test**
- run as / **maven install** (package + copy artifact in .m2/repository)

Le menu "**run as / maven build ...**" permet de faire apparaître une boîte de dialogue où l'on peut saisir plein de détails dont le nom du but/goal à atteindre (ex : *package*)



Il est souvent intéressant de donner un nom logique parlant à cette configuration (exemple : "**maven-package**") de façon à pouvoir relancer rapidement celle ci via le menu "**run as / maven build**" et une sélection de **configuration antérieure** .

NB : il est possible de retoucher certains paramètres via le menu "project / run configurations ..."

## 4.6. Diverses astuces "eclipse/maven" :

- Un "Maven 3" est intégré au plugin m2e pour eclipse et une installation préalable de maven n'est pas indispensable (bien que possible et paramétrable dans le workspace eclipse).
- Le plugin eclipse "m2e " nécessite absolument le jdk complet (plutôt que le JRE) pour fonctionner. Il faut donc pointer vers le répertoire "jdk ..." plutôt que "jre ..." au niveau du menu "**Windows préférences / Java / Installed JREs ...**"
- En cas de désynchronisation "maven-eclipse" , on peut tenter la séquence suivante :  
**run as / maven clean**  
**run as / maven install**  
**project / clean (eclipse)**  
**server / tomcat / clean** (si projet web)  
**run as / run on server** (si projet web)  
refresh navigateur (si projet web)
- En cas de bug inexplicable et non identifié , il peut quelquefois être utile de redémarrer entièrement eclipse (comportant quelquefois quelques bugs ou bien rendu instable suite à des manipulations erronées ) .
- Quelques fois , certains ".jar" sont mal téléchargés (mal recopiés , fichiers corrompus ) et il faut manuellement **supprimer certains sous répertoires de \$HOME/.m2/repository** en fonction des groupId/artifactId du message d'erreur maven .



## VI - JUnit et eclipse

### 1. Les principes des tests unitaires

#### 4 phases d'un test unitaire

1) **Initialisation** (méthode *setUp* ou *init*): définition d'un environnement de test complètement reproductible (une "fixture" avec par exemple une instance de composant bien initialisée et un éventuel jeux de données)

2) **Exécution du code à tester**: appel d'une méthode avec certains paramètres d'entrée bien choisis (valides ou invalides, ...).

3) **Vérification (assertions)**: comparaison du résultat obtenu avec la valeur de réponse attendue. Ces assertions définissent le résultat du test: SUCCÈS ou ÉCHEC .

4) **Désactivation/terminaison** (méthode *tearDown* ou ...): désinstallation des "fixtures" pour retrouver l'état initial du système, dans le but de ne pas polluer/perturber les tests suivants. Tous les tests doivent idéalement être indépendants et reproductibles.

NB: Selon le degré de sophistication/complexité du test unitaire, les phases 1 et 4 pourront être triviales ou très évoluées (ex : avec *dbUnit* ).

## Utilités/objectifs des tests unitaires

- **Bien appréhender/formaliser le contrat technico/fonctionnel d'un composant** logiciel (en lisant le code d'un test bien écrit , on comprend le comportement attendu des opérations/méthodes du composant à tester).
- **Trouver les erreurs rapidement et simplifier la maintenance :**  
Une fois le test unitaire écrit, on peut le relancer automatiquement sans effort des milliers de fois pour *vérifier l'absence de régression* et pour *localiser rapidement une erreur* en cas de problème .
- **Développer consciencieusement** (en testant naturellement l'absence de bug et le bon fonctionnement) au fur et a mesure de la programmation.
- S'assurer que tout le code écrit (et prévu pour être appelé/invoqué) soit **couvert** par un nombre suffisant de tests unitaires.
- Ne pas se contenter de la boutade "*tester c'est douter*" !

## Stratégie habituelle de test (Test Driven Development)

- 1) **définir la structure du composant à tester** (exemple: *modèle UML* , *interface java* , ...) et un éventuel embryon d'implémentation.
- 2) **écrire la classe de test** (en s'appuyant sur une structure connue et définie du composant à tester mais sur une implémentation qui n'est pas encore opérationnelle) .
- 3) **lancer une première fois les tests unitaires** et vérifier normalement que "sans code d'implémentation finalisé" les tests remontent bien des "échecs".
- 4) **écrire le code d'implémentation du composant à tester**. Relancer les tests qui devraient normalement réussir.
- 5) coder et tester des **variantes** au niveau des tests (paramètres d'entrées volontairement erronés , vérification de remontée d'exception , ...)
- 6) poursuivre de manière incrémentale et itérative (nouvelle méthode,...)



## 2. Tests unitaires avec JUnit (3 ou 4)

### 2.1. Présentation de JUnit

JUnit est un *framework* simple permettant d'effectuer des **tests** (unitaires , de non régression, ...) au cours d'un développement java . [ Projet Open source ---> <http://junit.sourceforge.net/> , <http://junit.org> ] . *JUnit est intégré au sein de l'IDE Eclipse* . JUnit existe en versions 3 et 4.

La version 4 utilise des **annotations** pour son paramétrage (**@Test** , **@Before** , ....)

### 2.2. Structure d'une classe de test (ancienne version JUnit3)

#### (Ancien) JUnit 3

**Conventions  
de noms sur  
méthodes**  
**setUp()**,  
**testXy()**  
et  
**tearDown()**

héritage

```
import junit.framework.TestCase;

public class CalculateurTest extends TestCase {
    private Calculateur c; //objet/composant à tester

    protected void setUp() { //initialisation du composant à tester
        c = new Calculateur(); //setUp() appelée avant chaque testXy()
    }

    public void testAdd() {
        assertEquals( c.add(5,6) , 11 , 0.000001 );
        // ou assertTrue(condition_a_vérifier) .
    }

    public void testMult() {
        assertEquals( c.mult(5,6) , 30 , 0.000001 );
    }

    protected void tearDown() {
        // éventuel code de terminaison réinitialisant certaines valeurs
        // d'un jeu de données (méthode facultative) }
    }
}
```

JUnit 3 est basée sur des conventions de nommage:

- La méthode **setUp()** sera appelée automatiquement pour initialiser les valeurs de certains objets qui seront ultérieurement utilisés au sein des tests.
- Chaque test correspond à une méthode de type "**testXxx()**" ne retournant rien (**void**) mais effectuant quelques assertions (**Assert.assertXxxx(...)** )
- On peut éventuellement programmer une méthode **tearDown()** qui sera alors appelée après chaque terminé (ex: pour ré-initialiser le contenu d'une base après).

## 2.3. Structure d'une classe de test ( version actuelle JUnit4 )

### JUnit4 (avec annotations)

Plus besoin  
d'hériter de  
TestCase  
mais  
**@Before**  
**@After**  
et  
**@Test**  
attendus

```
import org.junit.Assert;
import org.junit.Test; import org.junit.Before;

public class CalculateurTest
{ private Calculateur c;

  @Before /* comportement proche d'un constructeur par défaut*/
  public void initialisation(){
    c = new Calculateur(); // déclenché avant chaque @Test .
  }

  @Test
  public void testerAdd() {
    Assert.assertEquals( c.add(5,6) , 11 , 0.000001 );
    //ou Assert.assertTrue(5+6==11);
  }

  @Test
  public void testerMult() {
    Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
  }
}
```

*Méthode statique*

Il existe @Before , @After (potentiellement déclenchés plusieurs fois [avant/après chaque test]) et @BeforeClass , @AfterClass (pour initialiser des choses "static")

NB: il faut que **JUnit-4...jar** soit dans le classpath /

#### La démarche conseillée consiste à

- \* coder un embryon des classes à programmer (code incomplet)
- \* coder les tests (voir précédemment) et les déclencher une première fois (==> échecs normaux)
- \* programmer les traitements prévus
- \* ré-effectuer les tests (==> réussite ???)
- \* améliorer (peaufiner) le code
- ré-effectuer les tests ( ==> non régression ???) \* ...

## 2.4. Fonctionnement de JUnit

### Une instance de "Test JUnit" pour chaque test unitaire !!!

**La technologie JUnit (en version 3 ou 4) créer automatiquement une instance de la classe de test pour chaque méthode de test à déclencher.**

→ constructeurs , setUp() et méthodes préfixées par @Before seront donc potentiellement appelés plusieurs fois !!!!  
 → la notion d'ordre d'appel des méthodes est inexistante (non applicable) sur une classe de test JUnit.

Ceci permet d'obtenir des tests unitaires complètement indépendants mais ceci peut quelquefois engendrer certaines lenteurs ou lourdeurs.

Certains contextes (plutôt "stateless") peuvent se prêter à **des optimisations "static"** (**@BeforeClass** , **@AfterClass**) .

En combinant JUnit4 avec d'autres technologies (ex : SpringTest) , on peut également effectuer quelques optimisations (initialisations spéciales) au cas par cas selon le(s) framework(s) utilisé(s).

**Assert.assertNotNull()** et **Assert.fail("message")** peuvent être pratiques dans certains cas (exceptions non remontées, ...)

### Ordre des méthodes de test sur une classe JUnit4

Une classe de test **JUnit4** peut comporter plusieurs méthodes de test. Chacune de ces méthodes correspond à un **test unitaire** (censé être indépendant des autres) et donc par défaut , pour garantir une bonne isolation entre les différents tests unitaires :

- l'ordre des méthodes de tests déclenchées est non déterministe
- chaque méthode de test est exécutée avec une instance différente de la classe de test

**Dans certains cas rares et pointus**, on peut préférer **contrôler l'ordre des méthodes qui seront appelées**. Les tests seront alors un peu **moins "unitaires"** et un peu plus "liés".

Ceci peut s'effectuer de deux manières :

- \* avec l'annotation **@FixMethodOrder** que l'on trouve sur les versions récentes de JUnit4 .
- \* En écrivant une classe contrôlant le déclenchement d'une **série ordonnée de tests unitaires** (avec **@Suite**) .

## 2.5. @BeforeClass et "static" (optimisations)

### **JUnit4** (avec "*static*" et "*@BeforeClass*")

**@BeforeClass**

**@AfterClass**

attendus pour

gérer des

éléments

"static"

```
import org.junit.Assert;
import org.junit.Test; import org.junit.Before;

public class CalculateurTest
{ private static Calculateur c;

    @BeforeClass /* appelée une seule fois */
    public static void initialisation(){
        c = new Calculateur(); // initialisation "static".
    }

    @Test
    public void testerAdd() {
        Assert.assertEquals( c.add(5,6) , 11 , 0.000001 );
        //ou Assert.assertTrue(5+6==11);
    }

    @Test
    public void testerMult() {
        Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
    }
}
```

*Méthode statique*

## 2.6. Suite de tests (@Suite)

### Suite ordonnées de tests (JUnit4)

Quelques usages potentiels (tests ordonnés):

séquence Create/insert , select , update , select , delete , ...

variantes au niveau intégration continue :

- Suite\_tests\_essentiel (pour build rapides)
- Suite\_complete\_tests (pour build de nuit)

*exemple :*

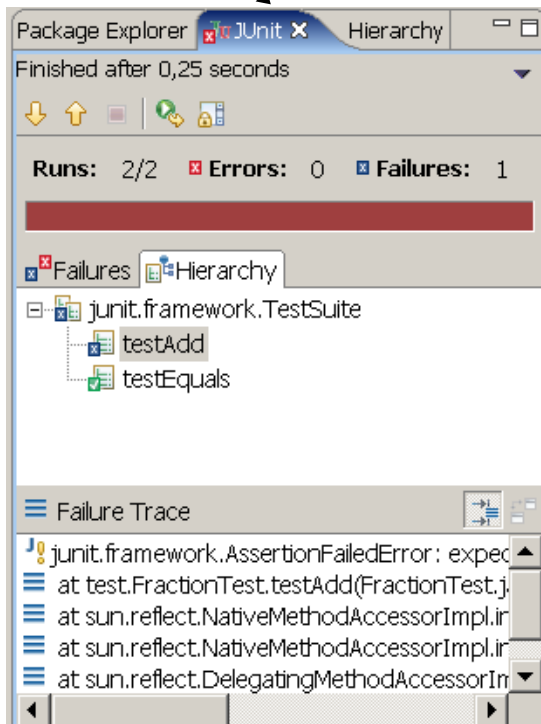
```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    JunitTest1.class,
    JunitTest2.class
})
public class JunitTestSuite {
}
```

## 2.7. Lancement des tests unitaires

### Lancement des tests unitaires

Depuis l'IDE eclipse:

**Run as ... / JUnit test**



TestSuite en JUnit3

et lancement après une sélection de **package** en JUnit4 pour lancer d'un coup toute une série de tests.

Comptabilisations :

**Error(s) :** exceptions java non rattrapées.

**Failure(s) :** assertions non vérifiées.

**VERT si aucune erreur.**

Depuis "maven" :

coder des classes nommées "**Test**Xy" ou "XY**Test**" dans **src/test/java** et lancement via **mvn test** ou autre.

## Combinaisons de frameworks (de tests) via **@RunWith**

```
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
...
// nécessite spring-test.jar et junit4.11.jar dans le classpath
@RunWith(SpringJUnit4ClassRunner.class)
// Chargement automatique de "/mySpringConf.xml" pour la configuration
@ContextConfiguration(locations={"/mySpringConf.xml"})
public class TestGestionComptes {

    // injection/initialisation du composant à tester contrôlée par @Autowired (de Spring)
    @Autowired //ou bien @Inject
    private InterfaceServiceXY serviceXy = null;

    @Test
    public void testTransferer() {
        serviceXy.transferer(...); Assert.assertTrue( ... );
    }
}
```

Il existe aussi **@RunWith(MockitoJUnitRunner.class)** et autres.

### 3. JUnit et Maven

Les classes des tests unitaires doivent être placées dans **src/test/java** .

Lancement des tests via maven :

**mvn test** ---> lance tous les tests dont les noms des classes commencent ou se terminent par "Test"

**mvn test -Dtest=XxxTest** -> lance que le test "XxxTest"

**mvn test -Dtest=\*xxTest** -> lance tous les tests finissant par "xxTest"

**NB:** par défaut , les tests unitaires sont systématiquement déclenchés lors d'un build ordinaire.  
L'option "**-DskipTests=true**" de mvn permet d'annuler/sauter l'exécution des tests.

# VII - GIT et eclipse

## 1. Présentation de GIT

**GIT** est un **système de gestion du code source** (avec prise en charge des différentes **versions**) qui fonctionne en **mode distribué**.

GIT est moins centralisé que SVN. Il existe deux niveaux de référentiel GIT (local et distant).  
Un référentiel GIT est plus compact qu'un référentiel SVN.

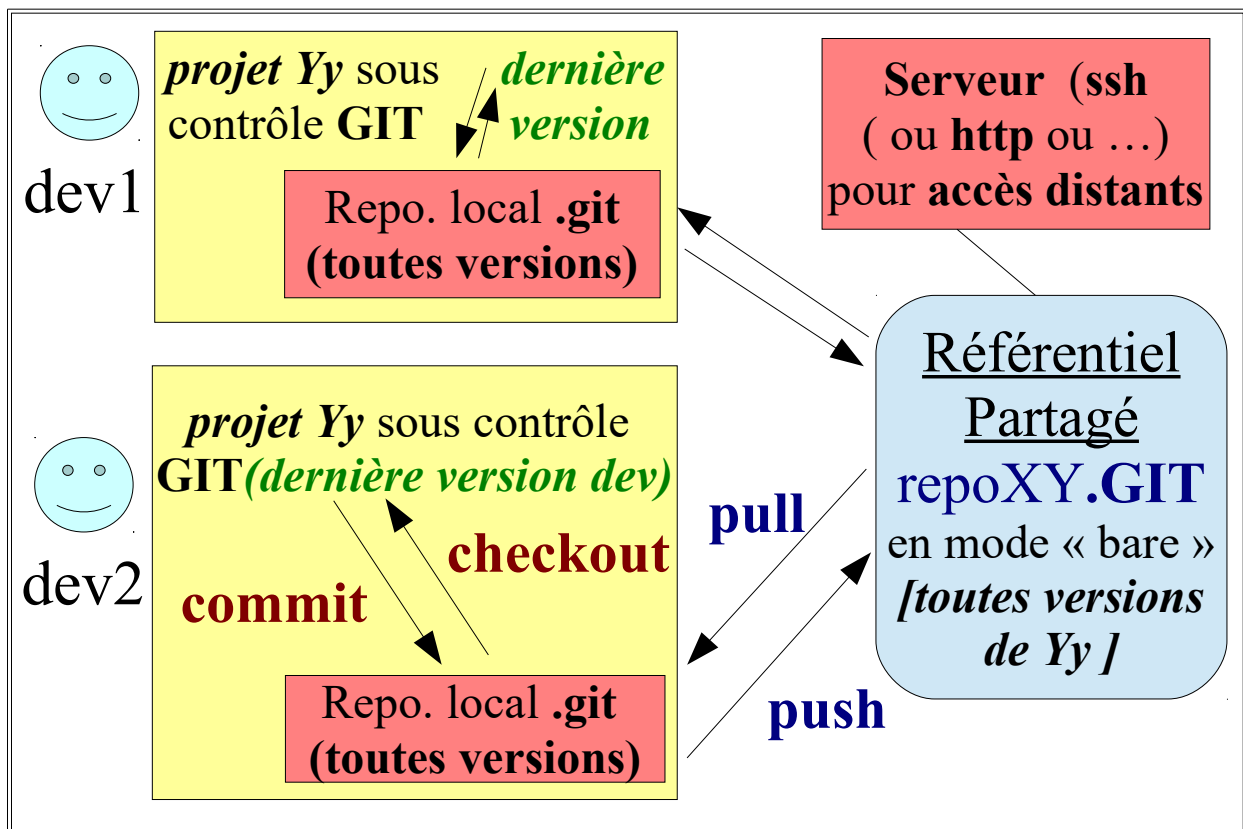
**GIT** a été conçu par **Linus Torvalds** (l'inventeur de **linux**).

Un produit concurrent de GIT s'appelle « **Mercurial** » et offre à peu près les mêmes fonctionnalités.

### 1.1. Mode distribué de GIT

Dans un système « scm » centralisé (tel que CVS ou SVN), le référentiel central comporte toutes les versions des fichiers et chaque développeur n'a (en général) sur son poste que les dernières versions des fichiers.

Dans un système « scm » distribué (tel que GIT ou Mercurial), le référentiel central ne sert que pour échanger les modifications et chaque développeur a (potentiellement) sur son poste toutes les versions des fichiers.





En bref, les commandes «**commit**» et «**checkout**» de **GIT** permettent de gérer le référentiel **local** (propre à un certain développeur) et les commandes «**push**» et «**pull**» de **GIT** permettent d'effectuer des **synchronisations** avec le **référentiel partagé distant**.

## 2. Configuration locale de GIT

Installation de GIT sous linux :

```
sudo apt-get install git-core
```

Configuration locale:

```
git config --global user.name "Nom Prénom"
git config --global user.email "poweruser@ici_ou_la.fr"
#...
```

# pour voir ce qui est configuré :

```
git config --list
```

## 3. Principales commandes de GIT (en mode local)

Commandes GIT (locales)	Utilités
<b>git init</b>	Initialise un référentiel local git (sous répertoire caché « .git » ) au sein d'un projet neuf/originel.
<b>git clone</b> <i>url_referentiel_git</i>	Récupère une copie locale (sous le contrôle de GIT et avec toutes les versions des fichiers) d'un référentiel git existant (souvent distant)
<b>git status</b>  <b>git diff</b> <i>fichier</i>	Affiche la liste des fichiers avec des changements (pas encore enregistrés par un commit) et git diff affiche les détails (lignes en + ou -) dans un certain fichier.
<b>git add</b> <i>liste_de_fichiers</i>	Ajoute un répertoire ou un fichier dans la liste des éléments qui seront pris en charge par git (lors du prochain commit).
<b>git commit</b> <i>-m message [-a] ou [liste_fichiers]</i>	Enregistre les derniers fichiers modifiés dans le référentiel git local (ceux précisés ou tous ceux ajoutés par <i>add</i> et affichés par <i>status</i> si <i>-a</i> )
<b>git checkout</b> <i>liste_de_fichiers</i>	Récupère les dernières versions depuis le référentiel local (sorte d'équivalent local du update de SVN , utile après un pull distant)
<b>git --help</b> <b>git cmde --help</b>	Obtention d'une aide (liste des commandes ou bien aide précise sur une commande)
<b>git log --stat</b> ou <b>git log -p</b>	Affiche l'historique des mises à jour

	-p : avec détails , --stat : résumé
<b>git branch</b> , <b>git checkout</b> <i>nomBranche</i> , <b>git merge</b>	Travailler (localement et ...) sur des branches
<b>git grep</b> <i>texte_a_rechercher</i>	Recherche la liste des fichiers contenant un texte
<b>git tag</b> <i>NomTag IdCommit</i>	Associer un <b>tag</b> parlant(ex: <i>v1.3</i> ) à un id de commit .
<b>git tag -l</b>	Visualiser la liste des tags existants
<b>git checkout tags/NomTag</b>	Récupère la version identifiée par un tag

**Exemples :**

#initialisation

cd p1; **git init**

#affichage des éléments non enregistrés

cd p1; **git status**

→ affiche:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#       modified:   src/f1.txt
#       modified:   src/f3-renamed.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

# commit all already tracked/added :

```
cd p1
# -a pour tous les fichiers listés dans git status
git commit -a -m "my commit message"
```

#commit all (with all new and deleted) :

```
cd p1
git add pom.xml.txt src/*
git status
# git commit gère tous les fichiers ajoutés (et supprimera de l'index ceux qui
# n'existent plus si option -a)
git commit -m "my commit message" -a
```

#checkout like local update

```
cd p1
git status
git checkout *
```

#historique des dernières mises à jour :

cd p1; **git log --stat**

---&gt; affiche:

```
commit 93446a0f2194089d83c941a63768f212eb96e0f8
Author: developpeur fou <moi@ici_ou_la.everywhere>
Date: Wed Dec 12 18:36:40 2012 +0100
    my commit message
pom.xml.txt      | 2 ++
src/fl.txt       | 1 +
src/f3-renamed.txt | 1 +
src/f4-renamed.txt | 1 +
src/p/pf2-renamed.txt | 2 ++
5 files changed, 7 insertions(+)
```

## 4. Commandes de GIT pour le mode distant

Commandes GIT (mode distant)	Utilités
<b>git init --bare</b>	Initialisation d'un nouveau référentiel vide de type «nu» ou «serveur». (à alimenter par un push depuis un projet originel)
<b>git clone --bare url_repo_existant</b>	Idem mais via un clonage d'un référentiel existant
<b>git clone url_repo_sur_serveur.git</b>	Création d'une copie du projet sur un poste de développement (c'est à ce moment qu'est mémorisée l'url du référentiel « serveur » pour les futurs push et pull)
<b>git pull</b>	Rapatrie les dernières mises à jour du serveur distant (de référence) vers le référentiel local. (NB: <i>git pull</i> revient à déclencher les deux sous commandes <i>git fetch</i> et <i>git merge</i> )
<b>git push</b>	Envoie les dernières mises à jour vers le serveur distant (de référence)  <u>Attention:</u> <i>le push est irréversible</i> et <i>personne ne doit avoir effectuer un push depuis votre dernier pull !</i>
...	

### Exemples:

#script de création d'un nouveau référentiel GIT (coté serveur) dans /var/scm/git ou ailleurs:

```
mkdir p0.git
cd p0.git
git init --bare
git update-server-info
mv hooks/post-update.sample hooks/post-update
```

```
#nb www-data est le groupe de apache2
cd ..
sudo chgrp -R www-data p0.git

# ce repository initial et vide pourra être alimenté par un push depuis un projet "original"
# depuis ce projet original , on pourra lancer git config remote.p0.url http://localhost/git/p0.git
#                               puis git push p0 master
echo "fin ?"; read fin
```

ou bien

```
# construira p1.git
git clone --bare file:///home/formation/Bureau/tp/tmp-test-git/original/p1
cd p1.git
git update-server-info
```

#récupération d'une copie du projet sur un poste de développement

```
git clone http://localhost/git/p1.git
```

#pull from serv:

```
cd p1
git pull
```

#push to serv:

```
cd p1
git push
```

## 5. Gestion des branches avec GIT

Tout projet commence avec une seule branche «**master**» .

Commandes GIT (branches)	Utilités
<b>git branch</b>	Affiche la liste des branches et précise la branche courante (*) .
<b>git branch</b> <i>nomNouvelleBranche</i>	Créer une nouvelle branche (qui n'est pas automatiquement la courante)
<b>git checkout</b> <i>nomBrancheExistante</i>	Changement de branche (avec mise à jour « checkout » des fichiers pour refléter le changement de branche) .
<b>git checkout</b> <i>master</i> <b>git merge</b> <i>autreBranche_a_fusionner</i>	Modifie la branche courante (ici «master») en fusionnant le contenu d'une autre branche
<b>git branch -d</b> <i>ancienneBrancheAsupprimer</i>	Supprime une ancienne branche (avec -d : vérification préalable fusion, avec -D : pas de verif , pour forcer la perte d'une branche morte)
...	

--	--

## 6. Gérer plusieurs référentiels distants

*s\_list\_remote\_git\_url.bat*

```
git remote -v
pause
```

*s\_set\_git\_remote\_origin.bat*

```
git remote set-url origin Z:\TP\tp_angular1.git
git remote -v
pause
```

*s\_push\_to\_remote\_origin.bat*

```
git push -u origin master
pause
```

*s\_push\_to\_github.bat*

```
git remote add GitHubMyContribOrigin https://github.com/didier-mycontrib/tp_angular.git
REM didier-mycontrib / gh14.....sm..x / didier@d-defrance.fr
git push -u GitHubMyContribOrigin master
pause
```

## 7. Plugin eclipse pour GIT (EGIT)

Le plugin eclipse pour GIT s'appelle EGIT .

### 7.1. Actions basiques (commit , checkout , pull , push)

→ Se laisser guider par la perspective "GIT" et via le menu "Team"

### 7.2. Résolution de conflits

- 1) déclencher "**Team / pull**" pour récupérer (en tâche de fond) la dernière version (partagée / de l'équipe). Le plugin EGIT va alors tenter un "**auto-merge**" ("git fetch FETCH\_HEAD" suivi par "git merge").
- 2) En cas de conflit (non résoluble automatiquement) , les fichiers en conflit seront marqués

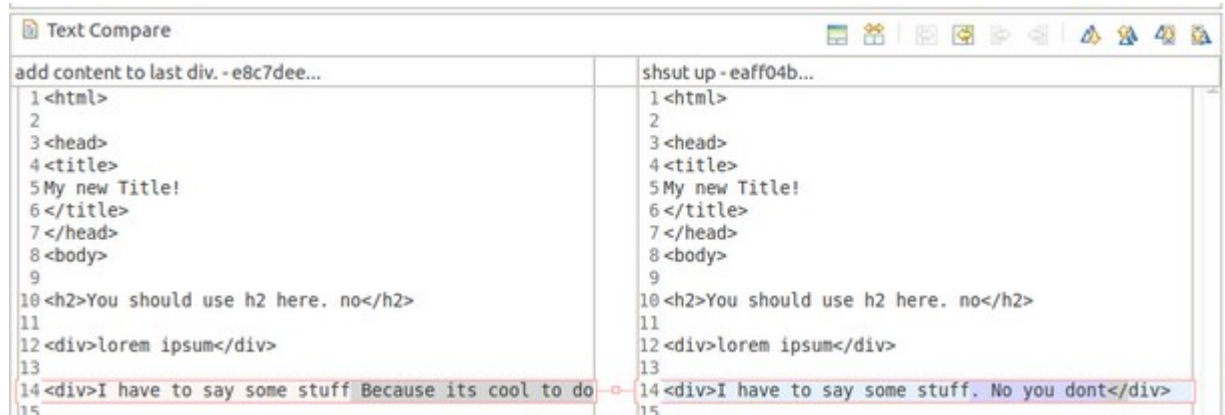
d'un point rouge.

```
<<<<<< HEAD
<div>I have to say some stuff Because its cool to do so.</div>
=====
<div>I have to say some stuff. No you dont</div>
>>>>>> branch 'master' of /var/data/merge-issue.git
```

Sur chacun des fichiers en conflit , on pourra déclencher le *menu contextuel*

**"Team / Merge tool"** . (laisser par défaut la configuration de "Merge Tool" : use HEAD).

- 3) **Saisir , changer ou supprimer alors au moins un caractère dans la zone locale (à gauche) + save :**



... au cas par cas ....

- 4) Déclencher le menu contextuel **"Team / add to index"** pour ajouter le fichier modifié dans la liste de ceux à gérer (staging).
- 5) Effectuer un **"Team / commit"** local .
- 6) Effectuer un **"Team / push to upstream ..."** pour mettre à jour le référentiel distant/partagé .

## VIII - Data-explorer et JDBC , Hibernate/JPA

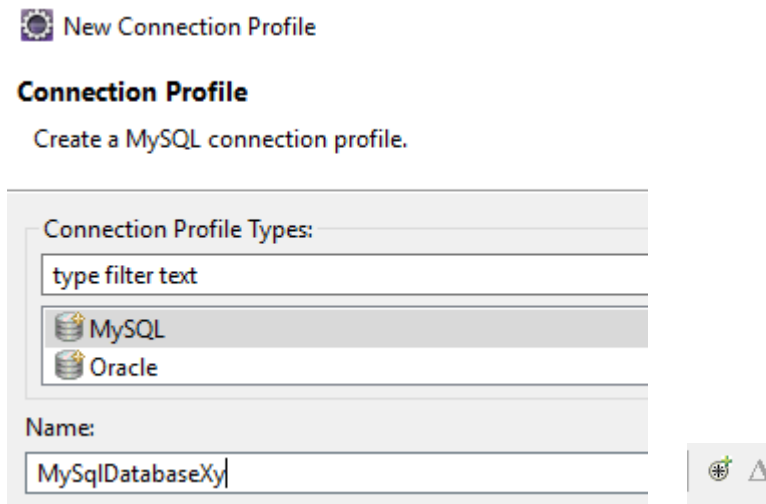
### 1. Eclipse et les bases de données

#### 1.1. DataSource-Explorer

**DataSource-explorer** est une vue d'eclipse qui permet de **visualiser la structure et le contenu d'une base de données relationnelle**.

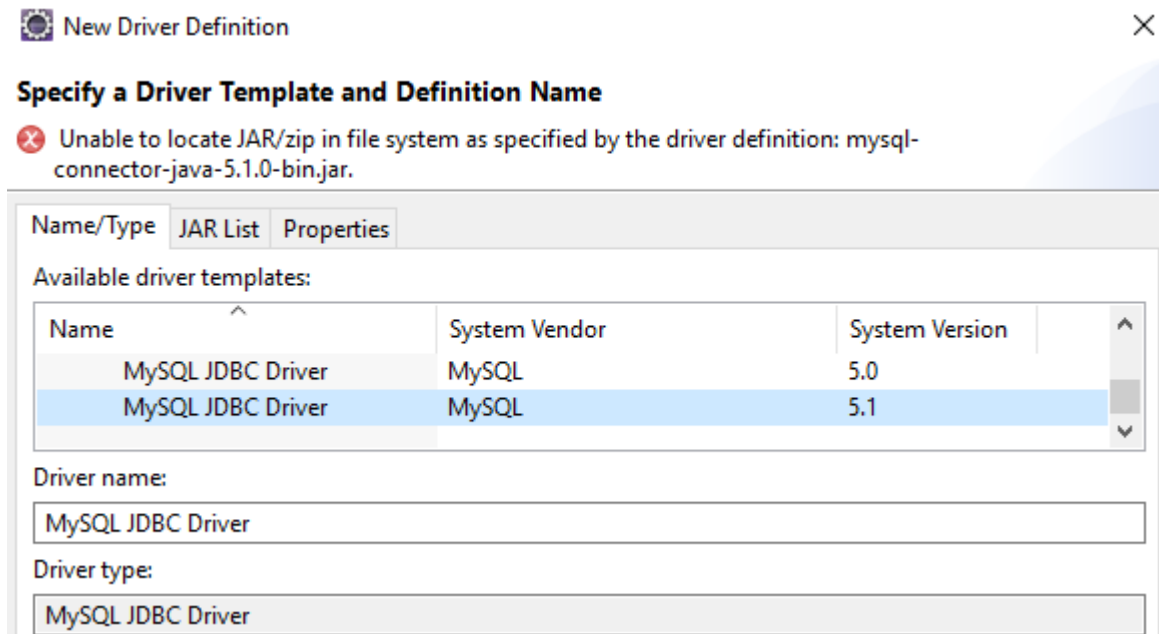
La vue dataSource-explorer s'affiche via le menu "Windows / **Show View** / Other ... / data management / **DataSourceExplorer** ).

Il faut d'abord paramétrer une connexion vers une base de données existante:

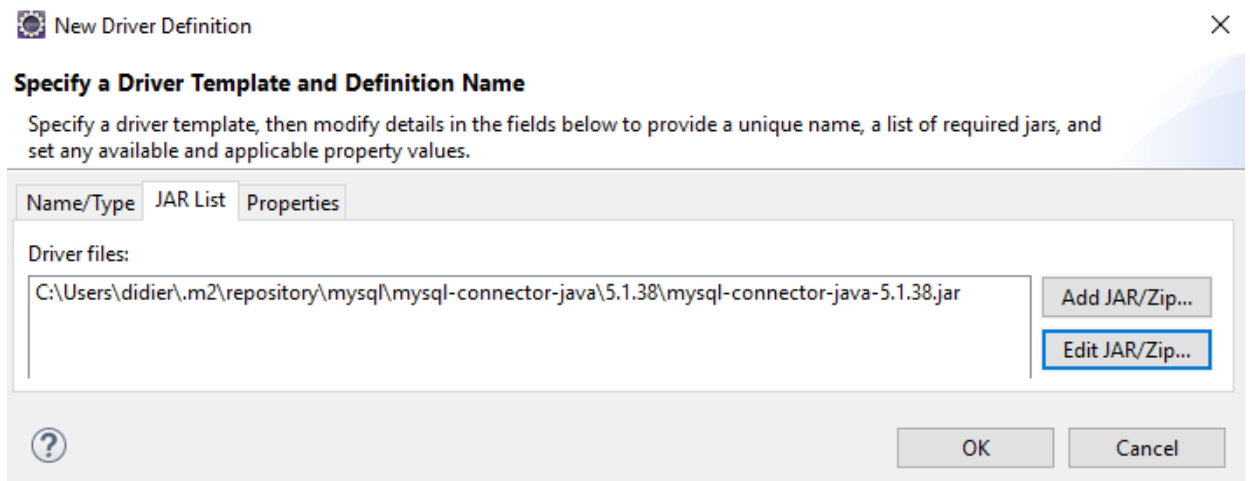


- il faut souvent commencer par choisir le type de base de données (ex : Mysql) et donner un nom logique à la connexion (ex : MySQLDataBase) .
- On a souvent besoin de paramétrer un **driver-jdbc** (selon le type de base de données : mysql , oracle , h2 , ...) :

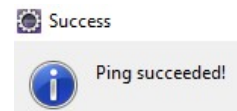
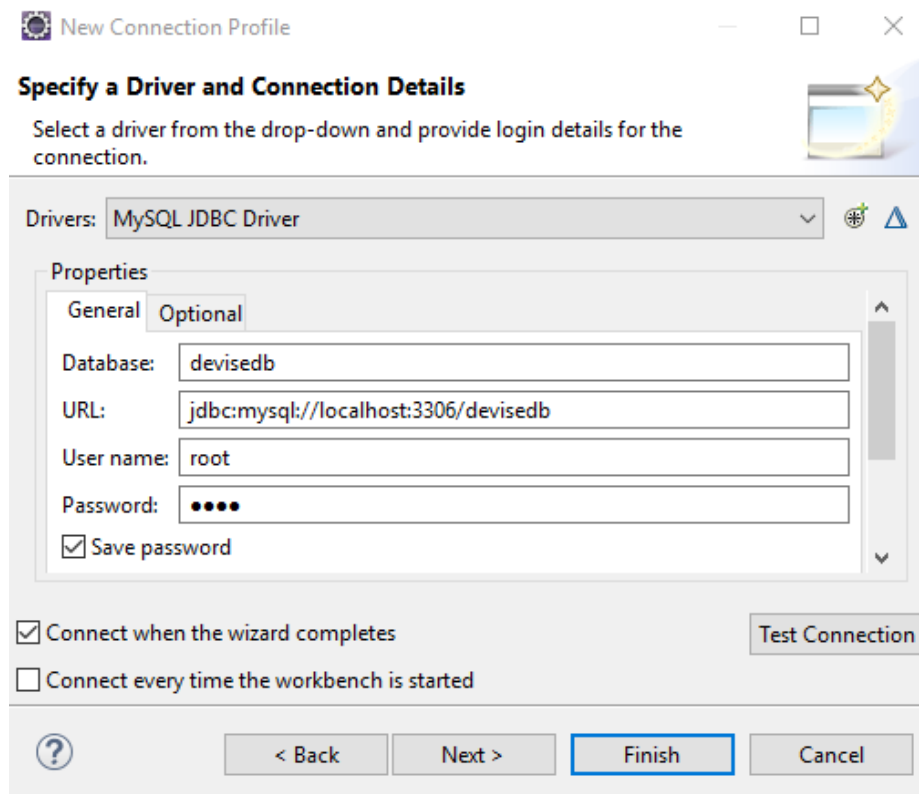




- De façon à préciser le chemin du ".jar" du driver JDBC (souvent situé dans \$HOME/.m2/repository) , il faut (dans l'onglet jar list) , indiquer le chemin adéquat :



- il faut ensuite paramétrer l'accès à une base de données spécifique ( nom logique , url , username, password , ...).



- on peut alors tester la connexion SQL via "Test Connection" .

#### Réglages "DataSource-explorer":

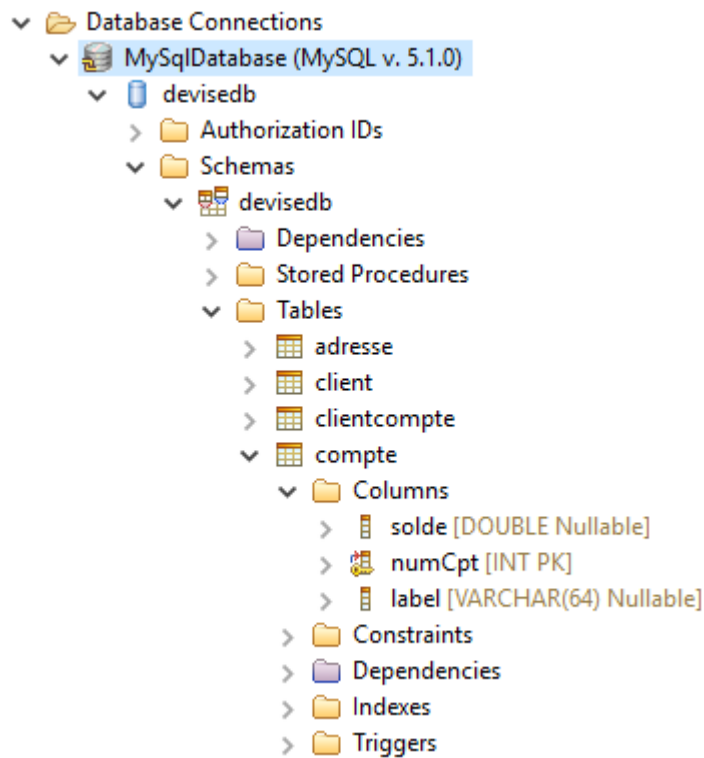
pour Oracle , corriger le nom du driver jdbc: oracle.jdbc.driver.OracleDriver

pour H2: selectionner Generic JDBC , org.h2.Driver , sa , ...

penser à régulièrement se connecter et se déconnecter de la base H2 pour éviter un blocage (verrou)

Une fois la connexion paramétrée et établie, on peut visualiser la structure et le contenu de la base de données de la façon suivante :

- On développe l'arborescence database / schéma / tables / tableXY (ce qui permet déjà de bien visualiser la structure : listes des colonnes avec noms et types )

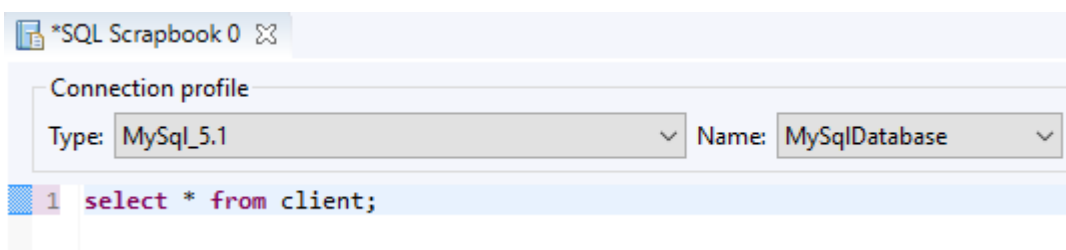


- Via le menu contextuel "**Data / Sample Content**", on peut récupérer un jeu de données (dans la table sélectionnée). Les données extraites sont alors affichées dans un autre onglet intitulé "**SQL Results**".

Status	Result1
	label numCpt solde
1	compte courant 1 949.0
2	compte codevi 2 301.0
3	compte 3 3 52.0

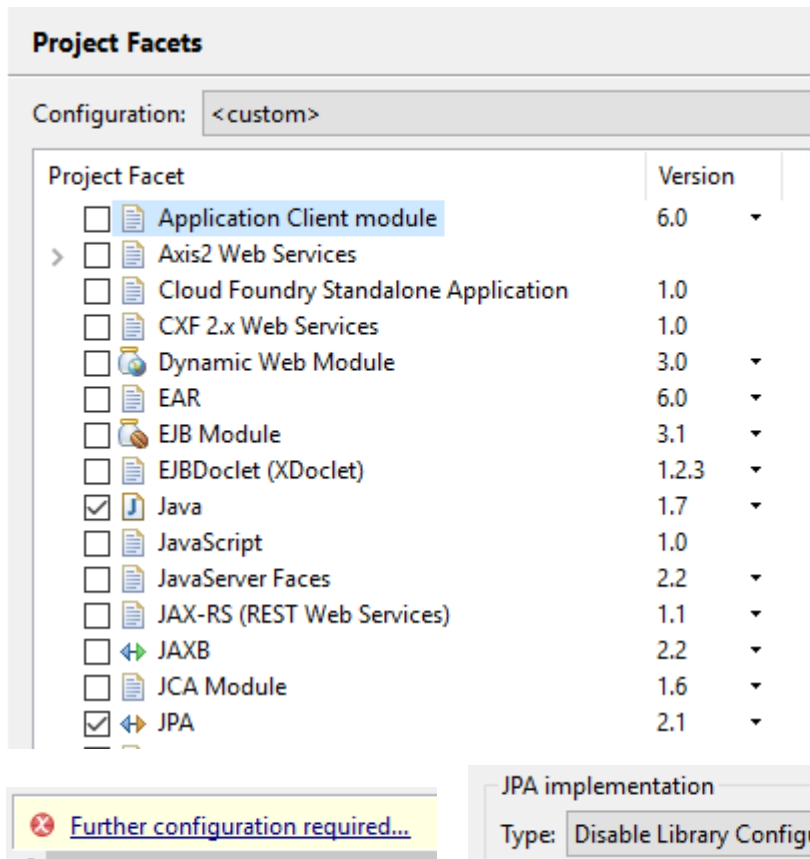
Pour lancer des requêtes SQL , on peut se placer sur la connection JDBC/SQL de DataSource-Explorer puis activer le menu contextuel "**Open SQL Scrapbook**".

Après avoir sélectionné le nom de la base et saisi la requête SQL il faut alors activer le menu contextuel "**Execute All**" pour obtenir le résultat dans la vue "SQL Results"



## 1.2. Eclipse et JPA (compatible "Hibernate")

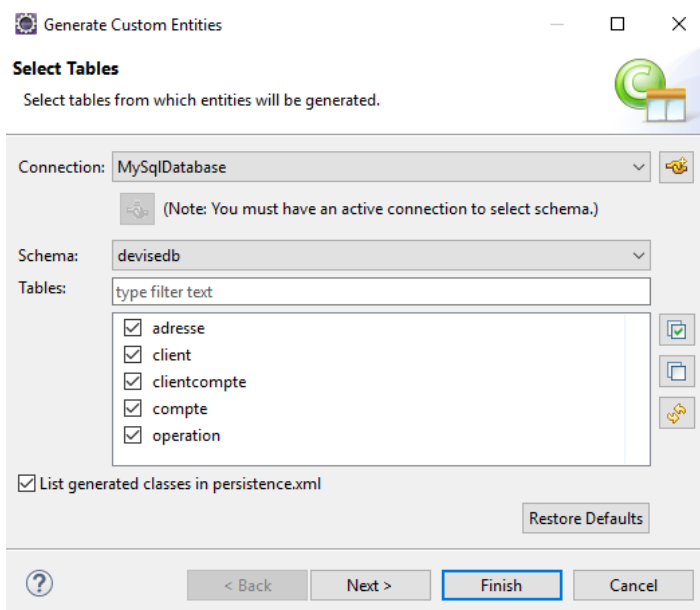
Pour bien exploiter les assistants "JPA" d'eclipse "Java EE" il faut commencer par ajouter (si nécessaire) la *facette "JPA"* à un projet java existant (ex : "maven") :




*NB.: pour une base H2 : redéfinir catalog = nom de la base et SCHEMA = PUBLIC*

Depuis ce projet ayant la facette JPA et en perspective "Java" ou "JEE" , déclencher le menu contextuel **"JPA Tools ..." / "Generate Entities from Tables"**.

Choisir la connexion SQL (préalablement paramétrée dans la vue "dataSource explorer"), le schéma de la base de données , les tables à prendre en comptes et quelques options (package java , ...)











 Generate Custom Entities

**Table Associations**

Edit a table association by selecting it and modifying the controls in the editing panel.

Table associations

 client	* ← 1 →	 adresse	 
Each adresse has many client.			
 compte	* ← *	 client	
Each compte has many client, and each client has many compte.			
 operation	* ← 1 →	 compte	
Each compte has many operation.			


☒ Generate this association

Cardinality: many-to-one

Table join: operation.ref\_compte=compte.numCpt


☒ Generate a reference to compte in operation


Property: compte


Cascade: 

☒ Generate a reference to a collection of operation in compte

Property: operations

Cascade: 

 < Back Next > Finish Cancel

 Generate Custom Entities

**Customize Defaults**

Optionally customize aspects of entities that will be generated by default from database tables. A Java package should be specified.

**Mapping defaults**

Key generator:

Sequence name:

You can use the patterns \$table and/or \$pk in the sequence name. These patterns will be replaced by the table name and the primary key column name when a table mapping is generated.

Entity access: ☒ Field ☐ Property

Associations fetch: ☒ Default ☐ Eager ☐ Lazy

Collection properties type: ☐ java.util.Set ☒ java.util.List

☐ Always generate optional JPA annotations and DDL parameters


**Domain java class**

Source folder:

Package:

Superclass:

Interfaces:



==> résultats = classes d'entités persistantes avec des annotations JPA (@Entity , @Table , @Id , @OneToMany , ....).

# IX - Configuration eclipse (plugin, ....)

## 1. Configurations (avancées) eclipse

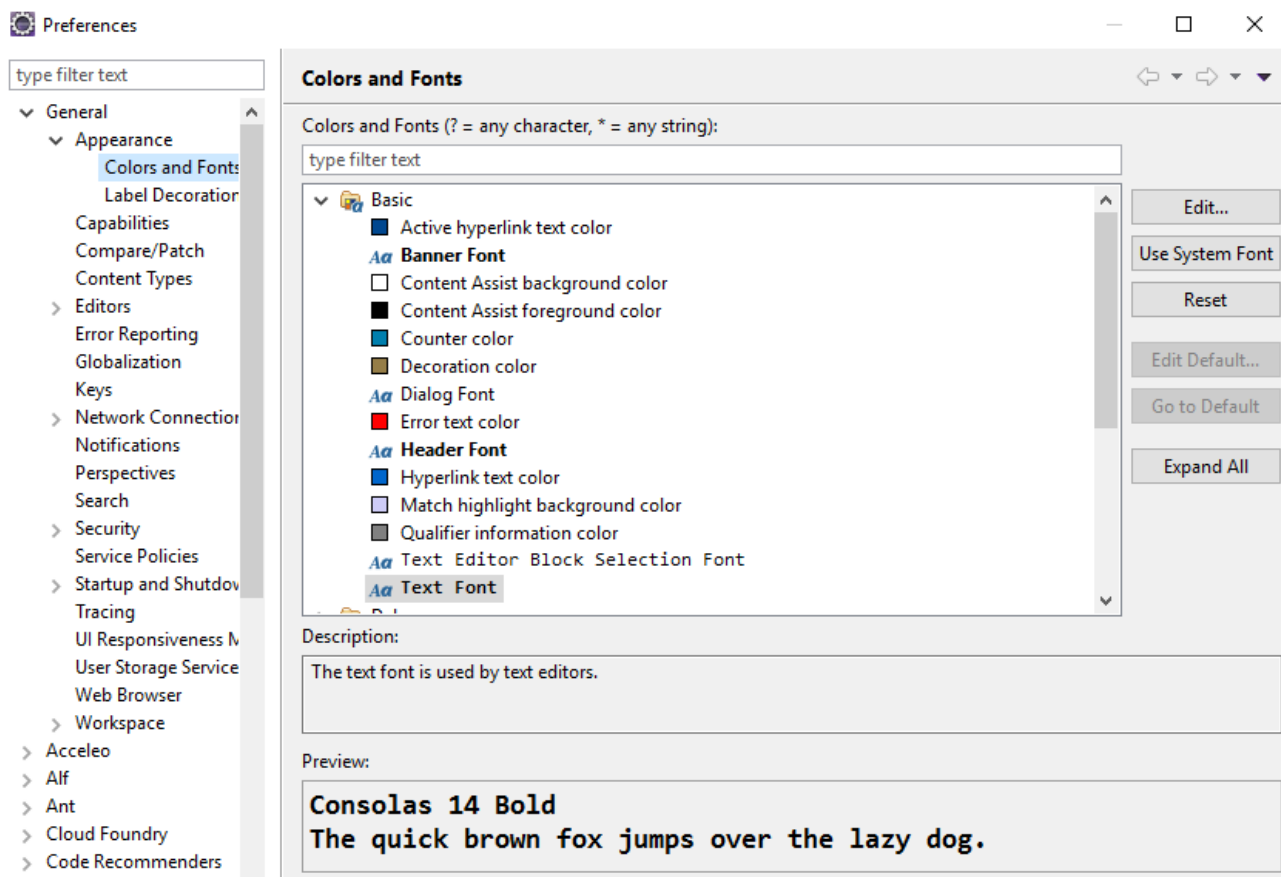
### 1.1. Configuration de la taille des caractères

Il est souvent nécessaire d'ajuster la police et la taille des caractères pour bien afficher le code au sein des éditeurs. Pour cela, le menu à activer est :

**Windows / Preferences ...**

**General / Appearance / Colors and Fonts**

**Basic / Text Font**     *Edit ..*



# ANNEXES



## X - Annexe – Liste des Travaux Pratiques

### 1. TP "prise en main d'eclipse"

Création de projets, manipulation générale de ressources...

### 2. Tp "Developpement java avec eclipse"

Développement d'une application minimale faisant appel à l'ensemble des fonctions offertes par l'environnement.

### 3. Tp "debug"

Débogage d'une application.

Débogage d'une application web.

### 4. Tp "maven et eclipse"

Build d'une application Web avec le plug-in Maven m2e.

### 5. Tp "JUnit et eclipse"

Mise en place de tests unitaires avec JUnit.