1 Familiarisation avec Maven (Hello World !)

Dans cette partie, Il vous est demandé de créer un premier projet en utilisant Maven afin de commencer à maîtriser ses bases.

1.1 Prérequis

Afin de réussir ce TP, les programmes suivants sont nécessaires:

1. **Java Développement Kit (JDK)**: l'outil de base qui vous permettra de compiler et de déboguer vos codes sources. Il existe plusieurs versions et plusieurs distributions différentes.

   Versions de la spécification: 8, 9, 11, 13, 14.

   Implémentation: Oracle JDK, OpenJDK, Amazon Corretto, etc.

   Recommandation: Utiliser la spécification 11 et l'implémentation OpenJdk. Cette implémentation open-source est assez stable, et ne contient pas de bibliothèques third-partie spécifiques

   Vous pouvez vérifier votre installation en exécutant la commande

   ```
   java --version
   ```

2. **Maven**: C'est l'outil de gestion que nous allons utiliser dans tous les TPs de ce cours. C'est un prédécesseur du *Gradle* que vous avez l'habitude d'utiliser en Programmation Android. Son utilisation vous paraîtra compliquée au début, mais vous vous rendrez compte rapidement que c'est un outils très efficace, et ça place est primordiale dans un projet de développement en Java. En fait, c'est *Maven* qui se chargera de l'exécution de toutes les directives du JDK, du téléchargement et de la gestion des dépendances vers des bibliothèques externes (et plus encore).

   Vérifiez votre installation en exécutant la commande la

   ```
   mvn --version
   ```

   La version recommandée pour mieux fonctionner avec le Jdk 11 est la version 3.8

3. **Un IDE**: La plupart des environnements de travail permettent de simplifier le développement en java et prendre en compte Maven. Vous pouvez utiliser *vscode* ou *apachenetbeans*12.

4. **Une connexion Internet**: Maven est un outil qui s'occupe lui même de télécharger les dépendance des dépôt de sources dans le cloud. La connexion à Internet est donc nécessaire pour programmer dans un environnement géré par **Maven**.

1.2 Premier programme

Ce tutoriel vous permettra de créer un projet Java de rien jusqu'à l'exécution de votre projet en utilisant les commandes Maven.

1.2.1 Génération du projet

Ouvrez votre interpréteur de commande (*terminal* ou *cmd*) et créez un nouveau répertoire appelé "gl". et naviguez pour vous positionner à l'intérieur.

```
mkdir gl
cd gl
```

Ensuite Générez un nouveau projet en executant la commande suivante

```
mvn archetype:generate \
      -DarchetypeArtifactId=maven-archetype-quickstart \
      -DarchetypeVersion=1.4 \
      -DinteractiveMode=false \
      -DgroupId=org.emp.gl \
      -DartifactId=firsttp
```

Une fois vous avez exécuté cette commande vous obtiendrez l'arborescence suivante:

```
firsttp
   |-- pom.xml
   `-- src
      |-- main
      |   `-- java
      |       `-- org
      |           `-- emp
      |              `-- gl
      |                 `-- firsttp
      |                    `-- App.java
      `-- test
         `-- java
            `-- org
               `-- emp
                  `-- gl
                     `-- firsttp
                        `-- AppTest.java
```

Jetez un œil au fichier *pom.xml*. Vous remarquerez qu'il contient beaucoup de configurations. Toutefois, pour l'instant nous ne nous intéresserons qu'à la partie décrivant l'**identité** de votre projet. voir le balises ci-dessous:

```
<groupId>org.emp.gl</groupId>
<artifactId>firsttp</artifactId>
<version>1.0-SNAPSHOT</version>
```

1.2.2 Édition du code

Maintenant ouvrez votre projet en utilisant l'éditeur de votre choix. Et éditez la classe **App** pour qu'elle vous affiche la chaîne de caractère "Bonjour GL". Contentez vous d'enregistrer votre source dans l'IDE (ne pas exécuter).

1.2.3 Compilation du code

Maintenant que notre code source est édité, nous devons procéder à la compilation grâce à la commande:

```
mvn compile
```

Remarquez la création d'un nouveau dossier **target** qui contient beaucoup de fichiers binaires "*.class*" (résultat de la compilation). Ces fichiers correspondent à votre code compilé, mais ne peuvent être exécutes.
Pour nettoyer tous ces fichiers auto-générés, il vous suffit d'exécuter la commande:

```
mvn clean
```

1.2.4 création de l'exécutable (archive)

Exécutez la commande:

```
mvn package
```

C'est une commande qui va exécuter la compilation, puis récupérera toutes les dépendances pour l'édition des liens, puis crée une archive pouvant s'exécuter dans différent environnement. L'archive peut être de type **jar** pour l'executer dans une *jvm* standard, de type **war** pour s'executer dans un serveur web, **nbm**, ...

Le type de l'archive souhaité est spécifié dans votre configuration **pom.xml**

1.2.5 exécution de l'archive générée

maintenant, afin d'exécuter l'archive générée, nous allons utiliser la commande Java :

```
java -jar target/first-tp-1.0-SNAPSHOT.jar
```

Vous allez remarquez que votre application ne fonctionne pas comme prévue. La raison est que votre *jvm* ne sais pas d'où commencer l'exécution de votre fichier binaire (point d'entrée). Ce dernier doit être spécifié dans votre configuration *pom.xml* . Faites la **configuration nécessaire** sur votre fichier *pom.xml*, puis recompilez et ré-exécutez.

Solution !

1.2.6 partage de l'archive

Jusque la, nous avons pu créer un module, nous l'avons archivé, puis exécuté. Maintenant, comment nous pourrions rendre ce module disponible pour qu'il soit utilisable par d'autres modules.

Maven prévoit ce cas de figure, et garde un dépôt local de tous les projets générés. Afin de rajouter votre module dans ce dépôt, il suffit d'exécuter la commande

```
mvn install
```

Vérifiez que votre *jar* est maintenant disponible dans le dépôt locale de Maven. .

habituellement, il se trouvera dans un répertoire caché nomé *.m2* qui se trouve dans votre dossier personnel. A l'intérieur, naviguez dans les répertoire selon les noms de packages que vous avez utilisez

2 Gestion de version

Maintenant que vous avez créé votre premier projet, nous souhaitons mettre en place un moyen qui nous permettrait de gérer l'évolution de notre projet et qui nous aiderait à garder une trace des modifications que nous effectuons sur notre code en fur et à mesure. Pour y arriver, il existe plusieurs gestionnaires de versions, tels que Mercurial (Hg), Subversion (svn) et Git. Pour nos TP, nous utiliserons ce dernier, vu sa popularité.

2.1 Étape 1 : Installation de GIT

Tout d'abords, allez sur le site web `http://git-scm.com`, télécharger et installez l'applicatif correspondant à votre système d'exploitation. Une fois installé, allez sur votre interpréteur de commande et testez son fonctionnement avec la commande

```
git --version
```

Si cette commande vous affiche un numéro de version, vous êtes prêts pour l'utilisation de votre gestionnaire de source. Mais avant cela, vous avez besoin de configurer deux variables nécessaire pour vous identité.

```
git config --global user.name "Mon Nom"
git config --global user.email "votreemail@votreemail.com"
```

2.2 Étape 2 : Utilisation de GIT

Maintenant que GIT est configuré sur votre périphérique Windows / Mac / Linux, nous allons explorer les bases de GIT. Et comment vous pouvez commencer à utiliser GIT.

2.2.1 Création / configuration / extraction d'un dépôt

Le dépôt est la chose la plus importante d'un projet. Pour transformer n'importe quel dossier en un dépôt GIT. on peut utiliser la commande simple

Pour votre cas ce cera la commande "*gitinit*." que vous executer à l'intérieur de votre répertoire de travail.

```
git init <dossier>
```

Un dossier caché nommé *.git* sera créé quand la commande aura été exécutée.

Maintenant qu'un dépôt a été mis en place, parlons de la structure de GIT. Chaque dépôt local se compose de trois "arbres" :

- le dossier de travail: qui contient les fichiers actuels;
- l'index: qui joue le rôle d'une zone de transit;
- le HEAD: qui est un pointeur vers le dernier commit effectué par l'utilisateur.

L'utilisateur ajoute un fichier ou des modifications à l'index (la zone de déploiement) et une fois revues, le fichier ou les modifications sont finalement confiées au HEAD.

2.2.2 Les commandes *add* et *commit*

Les modifications proposées ou les ajouts de fichiers sont ajoutés à l'index à l'aide de la commande *add*. Pour ajouter n'importe quel fichier à l'index, la commande est:

```
git add <nom_fichier>
```

Vous pouvez remplacer $< nom\_fichier >$ par un nom de dossier. Dans tel cas, l'ensemble des fichiers qui y sont contenus, seront ajoutés à l'indexe. par conséquent si vous exécutez la commande

```
git add .
```

Attention: avant d'exécuter cette commande, assurez vous de nettoyer le dossier de travail avec la commande *mvn clean*. Il est strictement recommandé de ne rajouter que les fichiers sources, et jamais de fichiers binaires

Tous les fichiers de votre répertoire de travail vont être rajoutés à l'index.

Si vous êtes assez confiant pour effectuer ces changements dans votre HEAD, vous pouvez utiliser la commande *commit*:

```
git commit --m \Message pour décrire le commit"
```

Une fois la commande *commit* exécutée (à partir du dossier de travail), les fichiers sont affectés au HEAD, mais il n'est toujours pas envoyé au dépôt distant.

Pour ce TP, nous nous contenterons de ces trois commandes, *init*, *add* et *commit*. La première vous permet de transformer votre répertoire de travail en dépôt, La seconde vous permet de rajouter des fichiers vers la liste des fichiers à observer. Et la dernière permet de créer un point de sauvegarde de votre code source que vous pouvez restaurer à tout moment.

Quand vous souhaiterez revenir vers une version ultérieur de votre code, il suffit d'appeler la commande

```
git checkout <identifiant du commit>
```

2.2.3 Autres commandes importantes

log, *tag*, *checkout*, $diff$, *remote*, *push*, *pull*, *stash*, *merge*, *branch*

Nous reviendrons sur ces commandes dans les prochains TP.

3 Logiciel Multi-modules

En utilisant $Maven$, vous pouvez regrouper plusieurs modules dans un seul projet. Chaque module à son propre fichier de configuration *pom.xml*. Tous les modules composant un projet se trouve dans le dossier ou est définit le projet *parent-project*.

Les modules sont organisés suivant une arborescence. Chaque module se situe dans le dossier du projet parent. Chaque module hérite toutes les configurations Maven du projet parent, et peut rajouter juste les configurations spécifiques dans son fichier de configuration *pom.xml*.

3.1 Le projet racine (*parent-project*)

Commencez par renommer votre projet *tp-racine* en utilisant dans le ficher *pom.xml* . En suite, rajouter la ligne suivante dans les propriétés de votre projet. juste au dessous de votre nom de projet

```
<packaging>pom</packaging>
```

En définissant le type de packaging comme *pom*, nous déclarons que le projet servira de parent ou d'*agrégateur*. Par conséquent, il ne produira pas d'exécutables. Du coup, le répertoire **src** ne nous servira à rien, donc, il pourra être supprimé.

Exécuter la commande

```
git commit --m "transformation du projet initial en un projet d'aggregation"
```

3.2 Les sous-modules

Maintenant, à l'intérieur du dossier généré, créez trois nouveaux modules avec la même commande de génération *mvn generate*. Appelons les comme suit:

1. lanceur:
   Ce module va contenir la class $Main$ qui se chargera du lancement de l'application. Dans cette application, nous souhaitons, gérer un personnel.

2. personnel-entities:
   Ce module contient une classe qui va définir l'entité Étudiant.

3. gestionnaire-du-personnel:
   Ce module contient une class, qui nous permet d'appliquer les différentes opérations possibles sur les entités. ex. $chercherEtudiant$, $ajouterEtudiant$, $sauvegarderEtudiant$

Avant d'écrire du code, remarquez que:

- dans les sous-modules, une balise *parent* est apparu pour indiquer le projet parent.

```
<parent>
    <artifactId>tp-racine</artifactId>
    <groupId>org.emp.gl</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
```

- dans le projet parent, une nouvelle balise *modules* apparut. Elle énumère tous les sous modules.

```
<modules>
    <module>lanceur</module>
    <module>personnel-entities</module>
    <module>personnel-manager</module>
</modules>
```

N'oubliez pas de faire un *commit*:

```
mvn clean
git add .
git commit -m "ajout des 3 sous modules vide"
```

3.3 Ajout des dépendance

Maintenant, implémentez différentes classes permettant de réaliser le projet.

Remarques

- Implémentez la classe *Etudiant*, dans le module personnel-entities, avec les attributs: "nom, prenom, et age". Ajoutez un constructeur, des getters et des setters.
- Implémentez le *GestionnaireEtudiant*, dans le module personnel-manager, une classe avec les méthodes sauvegarder, rechercher. Dans un premier temps, cette classe, va juste utiliser une liste d'Etudiant dans laquelle elle sauvegarde et elle recherche.

Vous allez remarque que la classe *GestionnaireEtudiant* n'arrive pas à reconaitre la class étudiant.

Pour régler ce problème, vous devez ajouter une dépendance dans le module personnel-manager vers le module personnel-entities, en éditant le fichier *pom.xml*, du module personnel-manager.

Solution ? Cliquez ici

3.4 Ajout d'un module pout l'affichage graphique

Ajouter un module pour l'affichage graphique. Pour ce module, au lieu de le créer manuellement, utilisez votre IDE. Si vous utilisez Netbeans, vous pouvez aller sur l'explorateur du projet, chercher le dossier modules, et avec un clique droit, ajouter un nouveau module.

Utilisez l'IDE pour la création d'une interface graphique qui vous permettra d'appeler le service de gestion du personnel, déjà implémenté, afin de simplifier l'interaction avec lui.