

## Applications monopostes II

Cégep Limoilou  
420-5B6-LI  
Automne 2023

Enseignant: Martin Simoneau

## Formatif 2a MAVEN

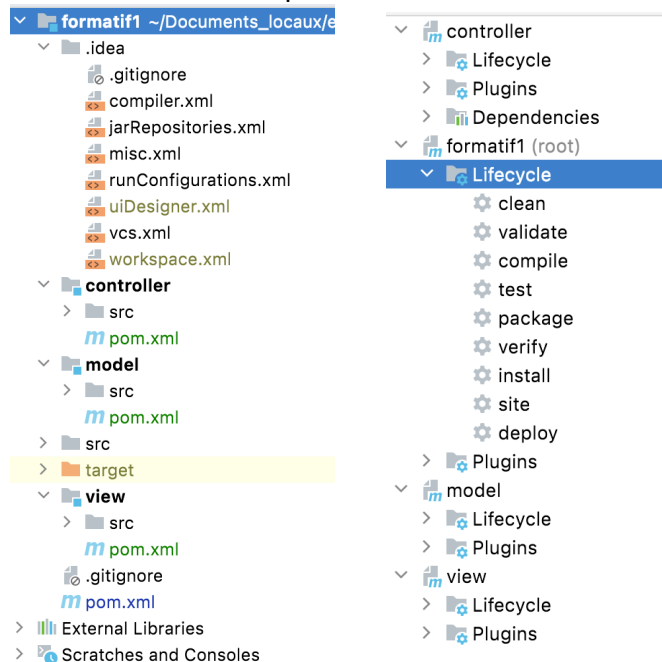
- **IntelliJ** modulaire avec **Maven**
- **Jar hell!**
- **JavaFX** multi-contrôleur

### Objectifs:

- Utiliser le gestionnaire de projet *Maven* avec *IntelliJ*.
- Créer un projet multimodules
- Utiliser des modules *jdk9+*
- Comprendre le *jar hell*

### À faire

1. Créer un dépôt *Git* qui servira pour tout le *formatif2a*
2. Dans *IntelliJ 2022.2*, créez un nouveau projet Maven dans le dépôt *formatif2a*.
  - Artificid et nom : ***formatif2a-maven***
  - Groupid : ***a23.climoilou.mono2.formatifs***
  - Version : ***0.1-SNAPSHOT***
3. Ajoutez un *.gitignore* pour ignorer le dossier *target*. Vous pouvez faire installer le dossier *target* avec *Maven*. Vous pouvez faire un premier commit du projet.
4. Création d'un MVC
  - Introduisez la classe *ApplicationComplete.java* qu'on vous a fournie. Mettez là dans le package *a23.climoilou.mono2.formatifs*.
  - Commitez
  - Créez 3 sous-modules Maven (controller, view et model) dans le projet *formatif1*. Vous obtiendrez la hiérarchie et le panneau Maven suivant :



- Assurez-vous que les fichiers *pom* des sous-module possèdent bien la balise **<parent>** et assurez-vous que le fichier *pom* du projet *formatif1* possède maintenant les balise **<modules>** pour chacun des sous-projets. *IntelliJ* a créé pour vous la structure Maven appropriée. Notez que les sous-modules peuvent tout de même être utilisés indépendamment les uns des autres.
- Transformez la classe *ApplicationComplete* en 3 classes : *Model*, *View* et *Controller*. Placez chacune des classes dans un package séparé portant le même nom que la classe dans le sous-module correspondant. Conservez la ligne commenté qui utilise un logger. Notez que le package de base de l'application devrait idéalement correspondre au *groupid* Maven.
  - Exemple : *package a23.climoilou.mono2.formatifs.controller*
- Ajoutez les dépendances maven nécessaires entre les sous-projets.
  - *controller* dépend de *view* et de *model*.
- Vérifiez que votre application fonctionne correctement et commitez.
- Effacez le code source du projet *formatif2a*, il est maintenant redondant. En fait, ce projet ne servira qu'à donner un *pom* commun aux sous-modules vous pouvez donc effacer les dossiers *src*, *target* au complet. **C'est important car un projet parent Maven ne doit pas contenir de code source!**

### 5. Jar Hell

Si l'on observe le code du modèle et le code de la vue on voit une ligne commentée qui utilise un objet *Logger*. Nous allons créer ce *logger* dans un quatrième module qui nous permettra d'expérimenter ce qui arrive lorsque 2 composants d'un même programme exigent des versions différentes d'un même module...

- Créer un nouveau module Maven dans le projet *formatif2a* :
  - Artificid et nom : **logger**
  - Groupid : **a23.climoilou.mono2.formatifs**
  - Version : **0.1-SNAPSHOT**
- Placez la classe *Logger.java* que vous avez reçue avec cet énoncé dans le nouveau module (attention de la mettre dans le bon package : *a22.climoilou.mono2.formatifs.logger*)
- Faites *Maven Install* sur le *logger* pour que les autres modules puissent le voir.
- Commitez
- Ajoutez les dépendances nécessaires dans le projet *model* et le projet *view* pour pouvoir utiliser le *Logger*.
- Utilisez le *logger*
  - Ajoutez un attribut *logger* dans la classe *Model.java* et dans la classe *View.java*.
  - Décommentez l'utilisation du *logger* dans les classes du modèle et de la vue. Si la méthode *log(String)* du *logger* utilise un deuxième paramètre, effacez-le. Il va revenir plus loin.
  - Vérifiez que le code fonctionne et que le *logger* écrit bien dans la console.
- Nous allons créer une nouvelle version du *logger* (0.2-SNAPSHOT)
  - Ajoutez un paramètre *boolean isError* dans la signature de la méthode *log*. *Log(String)* devient *Log(String,boolean)*

- Modifiez le code pour utiliser *System.err* au lieu de *System.out* lorsque le paramètre *isError* est *true*.
- Modifier la classe *model.Model* pour toujours utiliser le nouveau logger.
  - Changer la version de la dépendance *0.1-SNAPSHOT* -> *0.2-SNAPSHOT* dans le *pom* du modèle.
  - Appeler le *logger* avec le flag *isError* à *true* dans le modèle.
- Imaginons que les développeurs de la vue ne sont pas encore capables de passer à la version *0.2-SNAPSHOT*.
- Essayez d'exécuter le code avec le modèle qui utilise la version *0.2-SNAPSHOT* et la vue qui utilise *0.1-SNAPSHOT*. Vous obtiendrez :

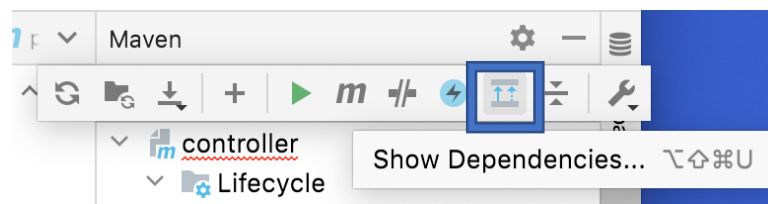
*Exception in thread "main" java.lang.NoSuchMethodError: 'void a23.climoilou.mono2.formatifs.logger.Logger.log(java.lang.String, boolean)'*

Cette erreur est due au fait qu'un projet java standard ne chargera pas 2 versions différentes d'une même classe. Le *Classloader* java ne peut pas le faire. Maven gère les versions, il peut même spécifier des numéros de versions plus précises (voir *version range references* dans :

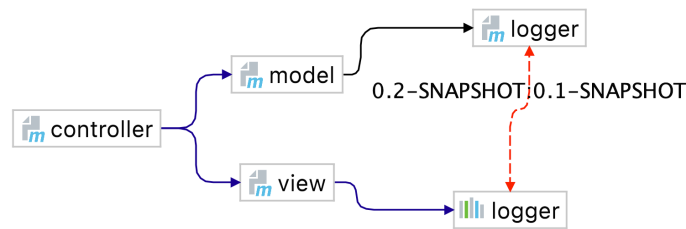
[https://docs.oracle.com/middleware/1212/core/MAVEN/maven\\_version.htm - MAVEN402](https://docs.oracle.com/middleware/1212/core/MAVEN/maven_version.htm-MAVEN402)), mais il ne modifie pas le *Classloader*. Ici , peu importe le numéro de version demandé, le *Classloader* ne pourra avoir qu'une seule version en mémoire à la fois (c'est une sorte de map). Par conséquent même si la version *0.1-SNAPSHOT* est chargée, la version *0.2-SNAPSHOT* va l'écraser parce qu'il s'agit de la même classe. Cette difficulté de pouvoir gérer précisément les numéros de version s'appelle le *jar Hell* et il existe des équivalents dans la majorité des langages ( *DLL hell* avec le C#). Certains logiciels comme le serveur *TomCat* et certains package comme le gestionnaire de module *OSGI* <http://docs.osgi.org/specification/osgi.core/7.0.0/ch01.html> (à la base de l'IDE *Eclipse*) gèrent les numéros de version en utilisant plusieurs *ClassLoader*. Mais leur usage dépasse ou ne cadre pas avec les objectifs de ce cours.

Concrètement, la façon courante de gérer le *jar hell* consiste à ne pas utiliser une version plus récente d'une librairie tant que **toutes** les dépendances transitives ne peuvent pas l'utiliser. Comment connaître ses dépendances transitives ?

1. Dans le panneau *maven*, sélectionnez le projet *controller* et appuyez sur le bouton **Show Dependencies**.



- Le graphique suivant devrait apparaître :



- *IntelliJ*, offre cet outil pour aider les développeurs à trouver les conflits de versions et pour les aider à bien choisir les modules qui sont compatibles. Notez qu'il s'agit d'un diagramme UML
- Pour corriger le problème, changer aussi la dépendance du projet *view* vers la version *0.2-SNAPSHOT* du *logger* et adaptez le code au besoin.

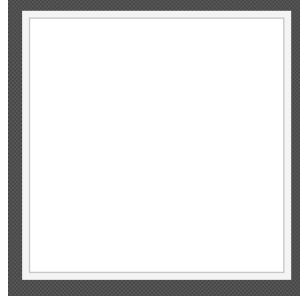
## 6. Utilisation de *JavaFX* 17

- Comme nous utilisons Azul jdk17 qui contient *javaFX*, nous n'avons rien de plus à faire pour utiliser *javaFX*.
- Nous allons maintenant améliorer notre application en changeant la console par un interface graphique *javaFX*.
- Nous allons ajouter les classe *ControllerFX*, *ApplicationFx* et *view.fxml* ainsi :
  - *view* : fichier  
`main/java/resources/a23/climoilou/mono2/formatifs/view/viewFX.fxml`  
 (interface graphique)
  - *model* : inchangé
  - *controller* : on ajoute les classes
    - `a23.climoilou.mono2.formatifs.controller.ControllerFX` ( le contrôleur *javaFX* lié au fichier *view.fxml*)
    - `a23.climoilou.mono2.formatifs.controller.ApplicationFX` ( l'application à lancer)
    -
- Remarquez que le *ApplicationFX* doit aller chercher *view.fxml* avec `"../view/viewFX.fxml"`. Cela signifie qu'on part de la classe *ControllerFx*, qu'on recule d'un package et qu'on entre ensuite dans le package *view* pour y trouver le fichier *view.fxml*.

## 7. Utilisation de *JavaFx* avec un multi-contrôleur

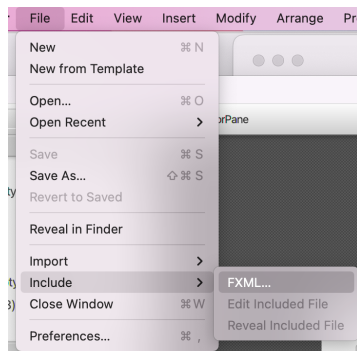
- Créer un fichier ***historique.fxml*** dans le package de ressources du module *view*.
- Ouvrez le fichier *historique.fxml* dans *SceneBuilder*. Effectuer les modifications suivantes:
  - La taille du *AnchorPane* root doit être 200x200.


- Ajoutez un contrôle *ListView* qui se tient à 5 pixels de chaque côté du root

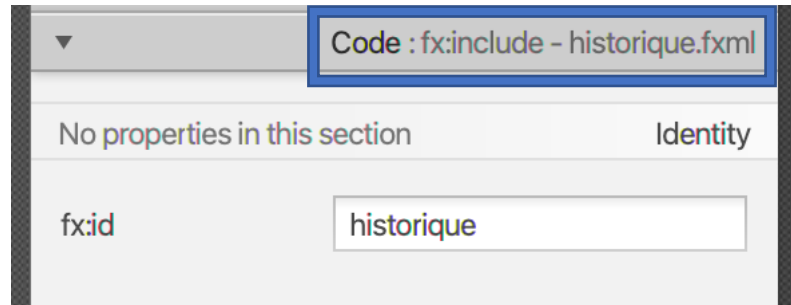


- associez au *ListView* un *FXID* : historique
- Associez-lui un *controller* (qu'on programmera plus tard)  
*a23.climoilou.mono2.formatifs.controller.HistoriqueController*
- Enregistrez les modifications
- Retournez dans *IntelliJ* et:
  - Créez la classe de controller *HistoriqueController*. Rappelez-vous que *SceneBuilder* peut vous aider à le faire (menu *view/ShowSample Controller Skeleton*)
    - Ajoutez une méthode *initialize()* avec l'annotation *@FXML*. Dans cette méthode ajoutez le texte "vide" dans le *ListView* historique.
    - Ajoutez une méthode publique *addItem(String item)* qui ajoute l'item reçu dans l'historique mais qui efface avant la chaîne "Vide" si elle s'y trouve encore.
- On veut maintenant inclure la *historique.fxml* directement dans le bas de *view.fxml*.
  - Ouvrez *view.fxml* dans *SceneBuilder* et:

- Inclure le fichier *FXML historique.fxml* à l'aide du menu



- Attachez ensuite le fichier inclus  *FXML AnchorPane* au root du fichier *view.fxml* à 5 pixels du bas de la droite et de la gauche.
- Donnez au fichier inclus le *FXID historique*. Assurez-vous que la mention *fx:include-historique.fxml* est écrit à côté de l'onglet *Code*.



- Grace à cet ID vous pourrez facilement accéder au contenu de la vue *include* à partir du contrôleur *JavaFX* parent.
- Enregistrez le tout!

○ Retournez dans *IntelliJ* et:

- Assurez vous que la ligne suivante est bien dans le fichier *view.fxml*

```
<fx:include fx:id="historique" source="historique.fxml"
AnchorPane.bottomAnchor="5.0" AnchorPane.leftAnchor="5.0"
AnchorPane.rightAnchor="5.0" />
```

- Retournez dans la classe *ControllerFX* et ajoutez les attributs:

```
@FXML
private AnchorPane historique;

@FXML
private HistoriqueController historiqueController;
```

- Le premier vous permet d'accéder au panneau racine de la sous vue historique. Normalement, il faut éviter de le faire car cette vue possède un contrôleur pour s'occuper d'elle.
- Le second va connecter le controller de la sous-vue (*historique.fxml*) au controller de la vue (*view.fxml*). Il sera ainsi possible d'interagir avec lui.
- **IMPORTANT: on ne peut pas donner n'importe quel nom de variable à ces dernières. Ici, il faut absolument utiliser le même mot qu'on a utilisé pour FXID de la vue incluse. "historique" pour la vue elle-même et "historique" suivi du suffixe "Controller" pour le contrôleur de cette vue.**
- On peut maintenant interagir avec le contrôleur et la vue imbriquée. Dans la méthode *ControllerFX.pushButton* ajoutez la ligne suivante

```
((ListView<String>)
historique.getChildren().get(0)).getItems().add(Integer.toString(value));
```

- Elle permet d'ajouter un item dans l'historique en passant par le panneau racine Java. Comme ce n'est pas très robuste (on présuppose qu'il est toujours à la position 0) on va plutôt interagir avec le contrôleur imbriqué et ce sera plus simple et plus robuste. Ajoutez la ligne suivante:

```
historiqueController.addItem(Integer.toString(value));
```

- On utilise simplement la méthode qui avait été programmée dans le contrôleur de l'historique.

FIN

---