

Introduction

L'objectif de ce TP est de vous faire développer une application JavaFX qui va intégrer les Expressions<E> ainsi que l'ExpressionParser<E> que nous avons développé lors du dernier TP. Néanmoins nous n'avons que 3 séances (étalées sur 2 demi-journées) pour ce faire. On se limitera donc à quelques aspects d'une application JavaFX sans rentrer en détail dans la conception.

Prérequis

Dans un premier temps téléchargez puis dézippez l'archive contenue dans /pub/TP4/TP-Expressions-JavaFX.zip, puis importez ce projet dans votre IDE préféré (voir aussi les [annexes](#) pour customiser votre IDE pour JavaFX).

Importation de vos expressions et parser du TP précédent.

Ce TP vous est fourni sous la forme d'un projet à compléter dans lequel il vous faudra importer ou copier/coller les classes que vous avez développées ou complétées lors du dernier TP dans ce nouveau projet. Voici les classes et des sous-packages du package expressions. Les classes soulignées devront être remplacées par votre implémentation. Les classes **en gras** sont nouvelles ou bien ont été modifiées par rapport au dernier TP.

- package expressions
 - AbstractExpression<E> est à remplacer par votre implémentation.
 - package binary
 - BinaryExpression<E> vous est fournie complétée.
 - AssignmentExpression<E> est à remplacer par votre implémentation.
 - AdditionExpression<E> vous est fournie complétée, vous pouvez ne pas l'importer.
 - SubtractionExpression<E> réalise la même opération que AdditionExpression<E>, il vous faudra donc importer votre implémentation ou customiser cette classe.
 - MultiplicationExpression<E> réalise la même opération que AdditionExpression<E>, il vous faudra donc importer votre implémentation ou customiser cette classe.
 - DivisionExpression<E> réalise la même opération que AdditionExpression<E>, il vous faudra donc importer votre implémentation ou customiser cette classe.
 - PowerExpression<E> réalise la même opération que AdditionExpression<E>, il vous faudra donc importer votre implémentation ou customiser cette classe.
 - **BinaryOperatorRules** a été modifié pour y ajouter une constante symbolique ANY représentant n'importe quel opérateur binaire et que l'on utilisera par la suite pour filtrer les expressions binaires.
 - package terminal
 - TerminalExpression<E> vous est fournie complétée.
 - ConstantExpression<E> est à remplacer par votre implémentation.

- **VariableExpression<E>** vous est fournie complétée et même modifiée. Il ne faut donc pas la remplacer.
 - Cette classe a été modifiée depuis le dernier TP afin que plusieurs instances d'une même variable "a" dans plusieurs expressions par exemple " $a + b$; $a = 2$ " puissent partager une seule et même valeur optionnelle : Ici la valeur 2. Cette classe contient donc maintenant un dictionnaire associant tous les noms de variables à leur valeur optionnelle :

```
private static Map<String, Optional<? extends Number>> values = new HashMap<>();
```

 Et donc la **valeur optionnelle** (`Optional<E> value;`) héritée de `TerminalExpression<E>` contenue dans chaque instance de `VariableExpression<E>` provient maintenant de ce dictionnaire ce qui permet de partager une même valeur entre plusieurs instances de `VariableExpression<E>`.
- **TerminalType** enum ajouté pour représenter symboliquement les différents types d'expressions terminales : constantes, variables, ou bien les deux.
- package `special`
 - **GroupExpression<E>** est une expression "virtuelle" (dans la mesure où elle ne réalise aucune opération arithmétique) qui ne fait que grouper plusieurs expressions ensembles. Nous nous en servons comme expression parente des toutes les expressions dans notre application.
- package `models` : Ce package contient tous les modèles (ou sous-modèles) de données relatifs aux expressions arithmétiques.
 - **ExpressionsModel<E>** sera notre **modèle de données principal** qui contiendra nos Expressions et les opérations à réaliser sur celles-ci: parsing, ajout, retrait, édition, etc.
 - **ExpressionTreeItem<E extends Number> extends TreeItem<Expression<E>>** est un nœud d'arbre (**TreeItem** au sens d'un **TreeView** JavaFX) qui contiendra l'expression racine (`GroupExpression<E>` vue ci-dessus) ainsi que toutes les expressions de notre application.
 - **NamedDataDisplay<E extends Number>** est une simple classe permettant d'associer un contenu texte et une valeur numérique à afficher ensembles dans un tableau (**TableView**) à deux colonnes (**TableColumn**). C'est la classe mère de deux classes :
 - **ExpressionDisplay<E extends Number> extends NamedDataDisplay<E>** qui sera utilisée pour afficher une expression et sa valeur dans un tableau.
 - **VariableDisplay<E extends Number> extends NamedDataDisplay<E>** qui sera utilisée pour afficher un nom de variable et sa valeur dans un tableau. (Voir `VariableExpression<E>` ci-dessus).
- package `parser`
 - **ExpressionParser<E>** est à compléter ou à remplacer par votre implémentation.

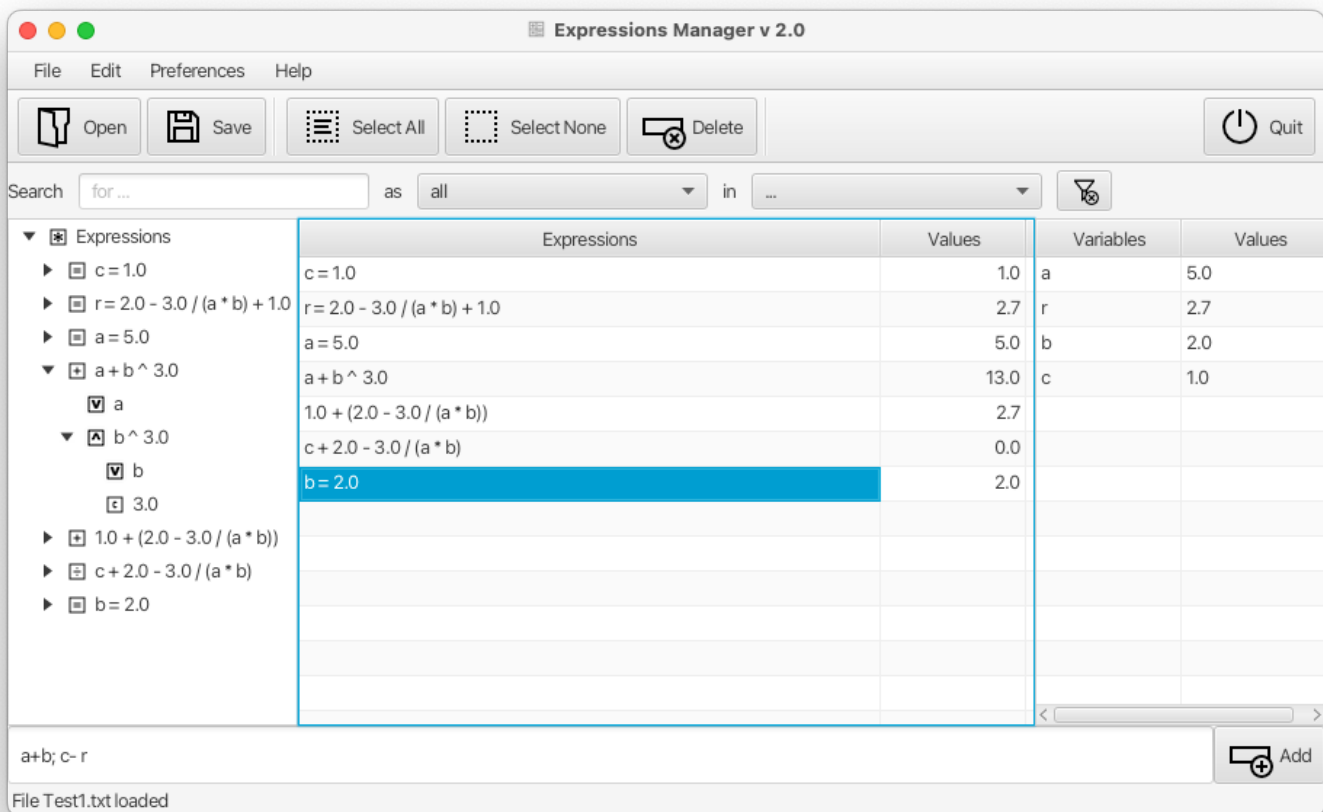
Importation de la librairie JavaFX

Ce projet utilise JavaFX pour la partie GUI (Graphical User Interface), il faudra donc configurer votre projet pour importer la bibliothèque JavaFX.

Voir les [annexes](#) pour l'installation et/ou l'utilisation de JavaFX dans votre projet.

L'interface graphique du projet

L'application JavaFX sur laquelle vous allez travailler se présente comme suit :



- La partie centrale de l'application est une table permettant d'afficher les expressions arithmétiques ainsi que leur valeur (si elles en ont une).
- La partie gauche de l'application est un arbre permettant d'afficher la structure des expressions arithmétiques.
- La partie droite de l'application est une table permettant d'afficher les variables partagées ainsi que leur valeur (si elles en ont une).
- La partie en bas de l'application permet de saisir une ou plusieurs expressions arithmétiques qui viendront alors s'ajouter aux expressions déjà présentes dans l'application.
- La partie en haut de l'application contient
 - Une barre de menu contenant les menus et items de menu des différentes actions de l'application (Ajouter une expression, charger un fichier, etc.)
 - Une barre d'outils qui contient des boutons permettant de déclencher les actions les plus courantes de l'application.
 - Ainsi qu'une barre de recherche permettant de filtrer les expressions suivant différents critères.

L'architecture du projet

L'architecture du projet suit de près le modèle MVC (**Model** / **View** / **Controller**) :

- La partie **Model** est représentée par la classe `expressions.models.ExpressionsModel`.
- La partie **View** est décrite dans le fichier FXML `application/ExpressionsFrame.fxml`.

- La partie **Controller** est représentée par la classe `application.Controller`.

Les éléments importants du projet sont les suivants :

- package `application`
 - `ExpressionsFrame.fxml` contient la description au format XML de l'interface graphique de l'application réalisée avec le logiciel **SceneBuilder** (voir l'[annexe](#)). Cette description correspond à la partie **Vue** dans le modèle MVC. Ce fichier FXML référence la classe `Controller` comme étant le contrôleur de cette interface graphique et nomme certains éléments de l'interface graphique que l'on aura besoin retrouver comme membres de la classe `Controller` : Il peut s'agir d'attributs comme de méthodes (des callbacks en l'occurrence).
 - `MainFX` contient le programme principal de l'application JavaFX. Ce programme principal charge le fichier `ExpressionsFrame.fxml` avec la méthode `load` d'un `FXMLLoader`. Ce chargement instancie le "contrôleur" de cette interface représenté par la classe `Controller`.
 - `Controller` représente le Contrôleur de notre application au sens du **Controller** dans l'architecture MVC.
 - Les deux éléments : le fichier `ExpressionsFrame.fxml` et la classe contrôleur `Controller.java` sont intimement liés : On retrouvera dans ce contrôleur :
 - Les attributs représentant certains éléments de l'interface graphique définis dans le fichier FXML avec lesquels on aura besoin d'interagir dans le contrôleur. Exemple : définissant un label dont on changera le texte dans le contrôleur : `messageLabel.setText("message");`

`@FXML`

```
private Label messageLabel;
```

- Les callbacks qui représentent les actions réalisées par notre application et déclenchés par des éléments de l'interface graphique. Exemple :

`@FXML`

```
public void onSelectNumberType(ActionEvent event)
{
    logger.info("Select Number type action triggered ...");
    ...
}
```

- Des attributs propres au contrôleur avec parmi ceux-ci une instance de la classe `ExpressionsModel<E>` qui représente notre modèle de données.

- package `expressions`
 - package `models`
 - `ExpressionsModel<E>` représente notre modèle de données au sens du **Model** dans l'architecture MVC. Il s'agit ici d'une liste d'`Expressions<E>`. C'est la classe responsable de l'implémentation concrète de toutes les opérations à réaliser dans notre application telles que :
 - Parsing et ajout de nouvelles expressions.
 - Lecture d'un fichier texte contenant des expressions.
 - Sauvegarde des expressions dans un fichier texte.
 - Sélection et suppressions d'expressions.
 - Edition d'expressions.

N.B. Tous les éléments annotés avec l'annotation `@FXML` dans la classe contrôleur doivent être référencés dans le fichier `ExpressionsFrame.fxml` soit au travers des noms des éléments (`fx:id`), soit au travers des callbacks à déclencher lors de l'interaction avec certains éléments.

- Extrait de ExpressionsFrame.fxml : Définition d'un bouton contenant le texte "Open" nommé openButton et qui déclenchera le callback onOpenAction dans le contrôleur lorsque l'on cliquera dessus.

```
...  
<Button fx:id="openButton" onAction="#onOpenAction" text="Open">  
...
```

- Extrait de Controller.java : Dans le contrôleur on doit donc retrouver ces deux éléments avec l'annotation @FXML

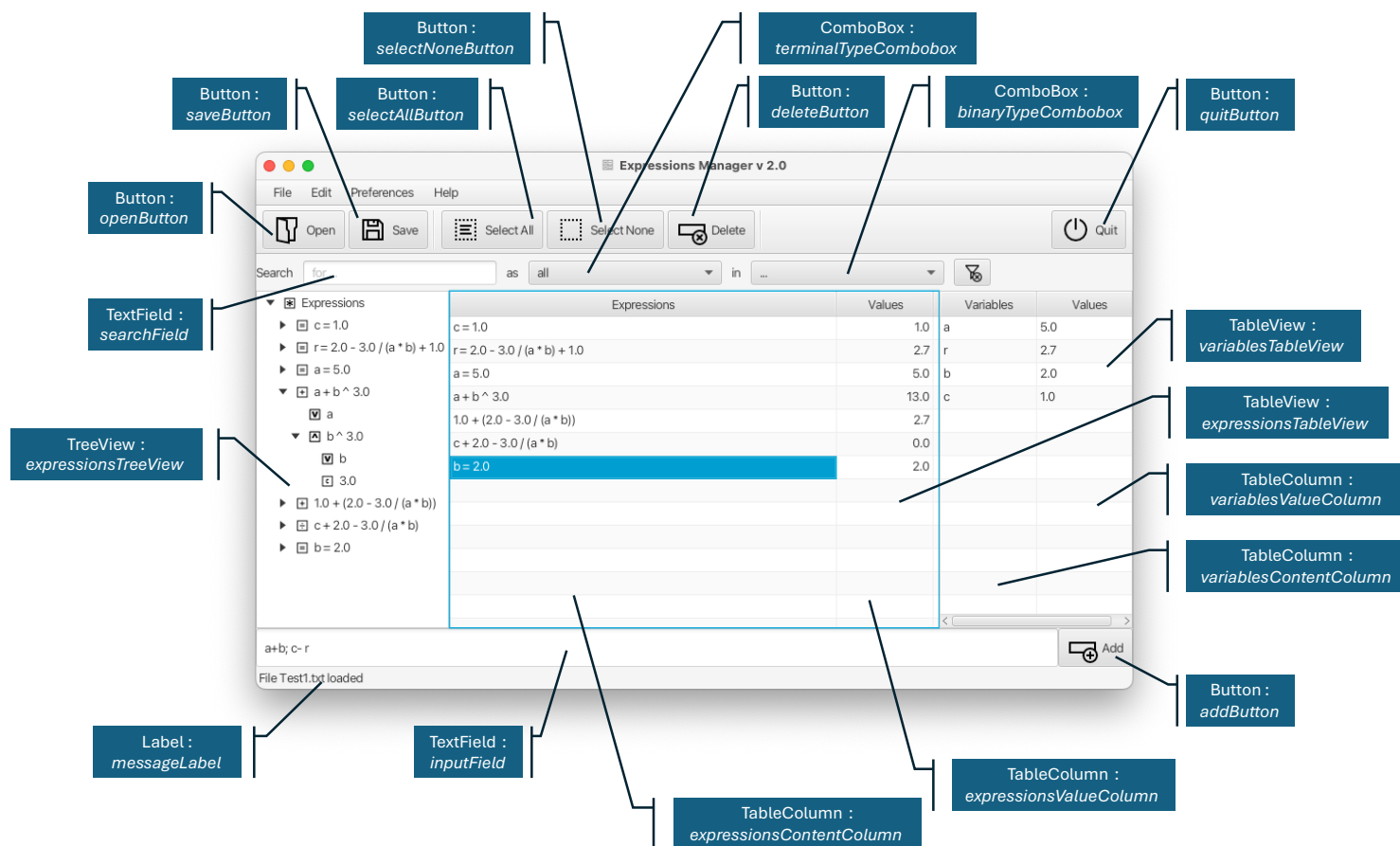
```
...  
@FXML  
private Button openButton;  
...  
@FXML  
public void onOpenAction(ActionEvent event)  
{  
    logger.info("Open Action triggered ...");  
    ...  
}  
...
```

Si un élément mentionné dans le fichier FXML n'a pas de pendant dans le contrôleur cela provoquera une erreur de chargement du fichier FXML dans la classe MainFX: SEVERE: Can't load FXML file ... suivi **parfois** d'une indication sur la nature de l'erreur. Si aucune indication n'est fournie il vous faudra debugger la méthode initialize du Controller pour déterminer ce qui ne va pas :

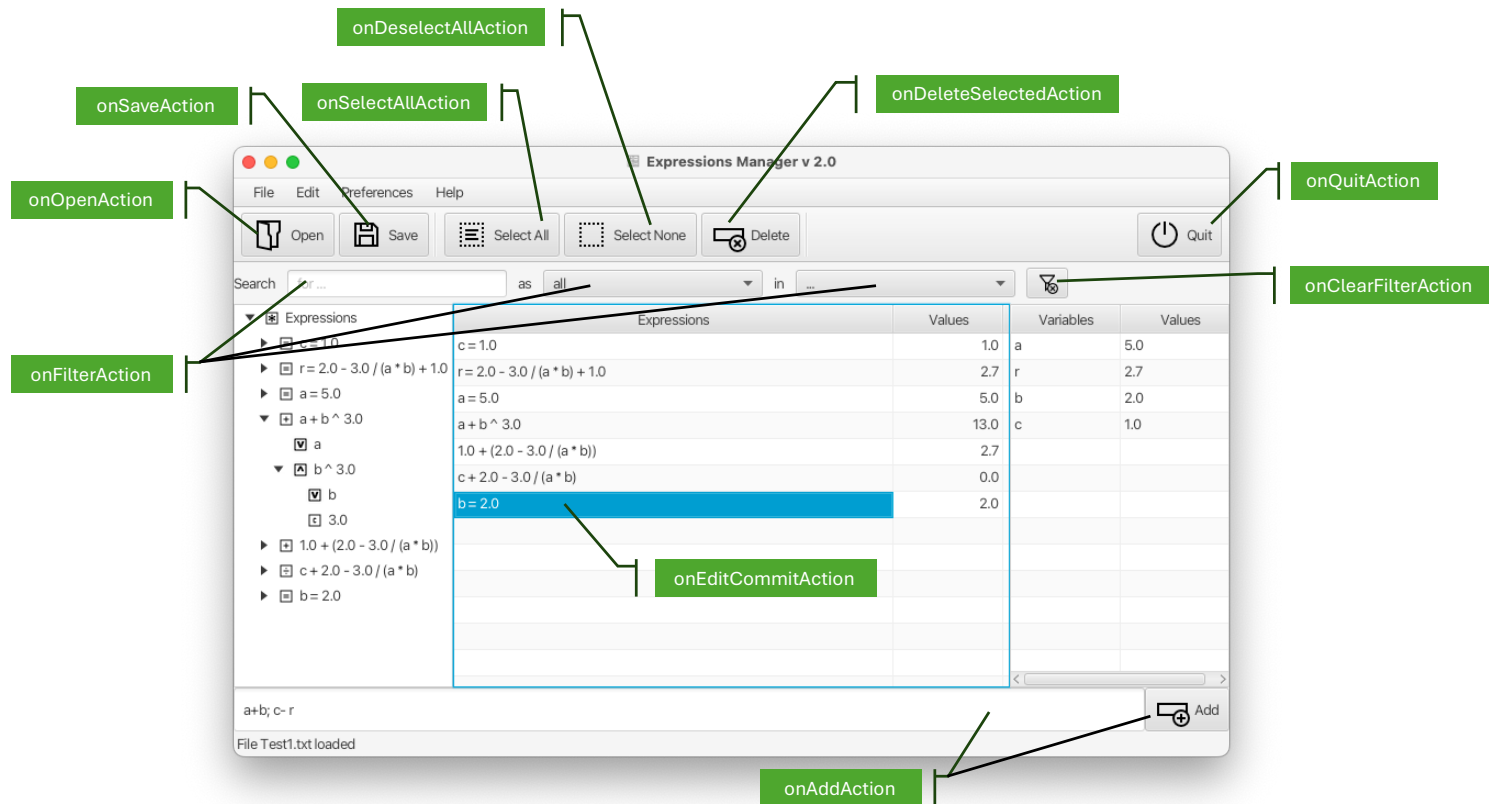
- Par exemple un attribut annoté @FXML dans le Controller qui n'est pas référencé dans le fichier FXML sera null car il n'aura pas été instancié par le FXMLLoader dans MainFX.

ExpressionFrame.fxml

La figure ci-dessous détaille les types et les noms des éléments d'interface graphique définis et nommés (avec un `fx:id`) dans le fichier `ExpressionFrame.fxml` et que l'on devra donc retrouver en tant qu'attributs annoté avec `@FXML` dans la classe `Controller` :

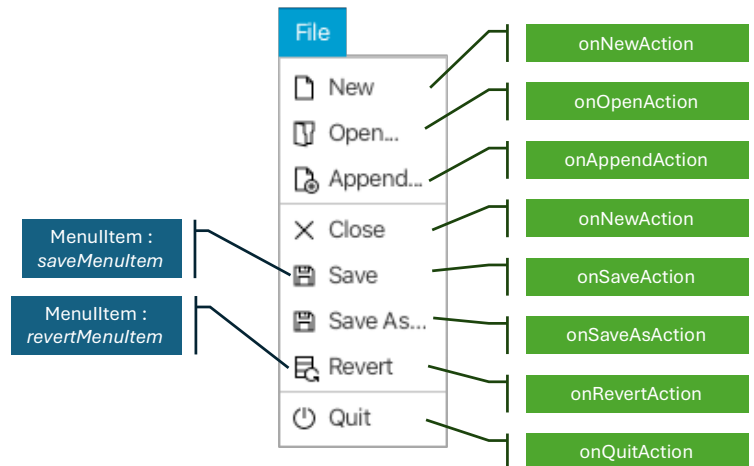


La figure ci-dessous détaille les différents callbacks du Controller associés aux éléments d'interface.

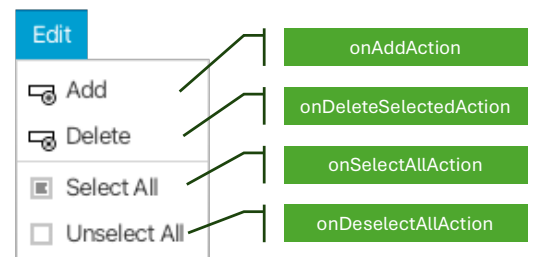


Les figures suivantes détaillent les types, noms et callbacks associés pour les divers éléments de menu.

- Pour le menu "File" :
Les items de menu `savedMenuItem`, `revertMenuItem` ainsi que le bouton `saveButton` sont nommés afin qu'ils puissent être désactivé (propriété `disabled`) lorsque le modèle de données ne possède pas de fichier vers lequel sauver les expressions.

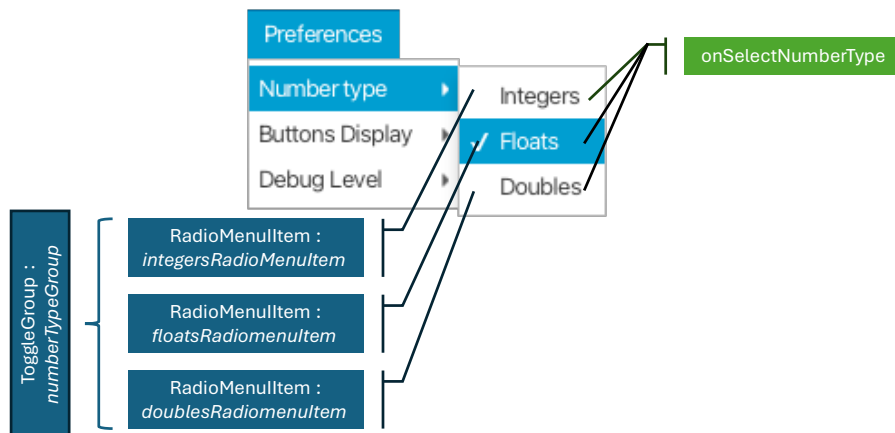


- Pour le menu "Edit" :

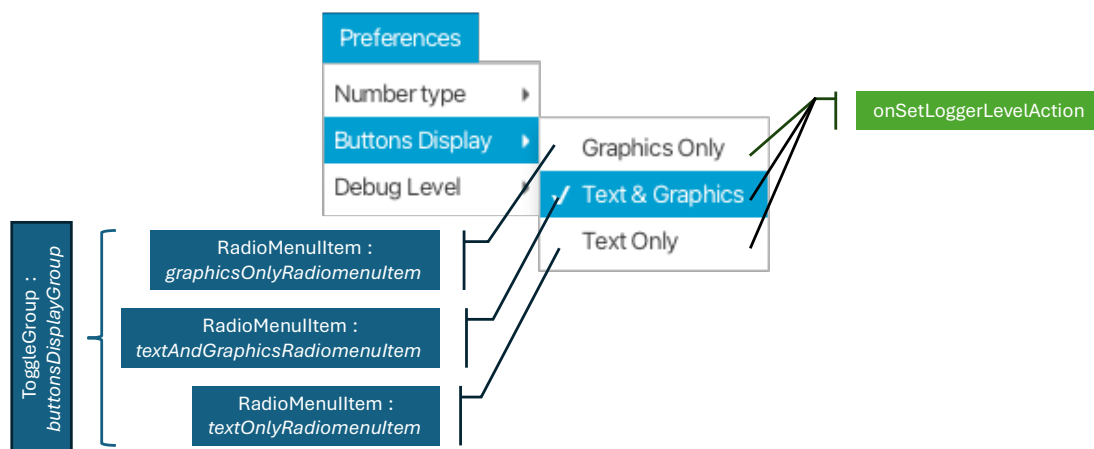


- Pour le menu "Preferences", sous-menu "Number type" :
Les 3 RadioMenuItem sont nommés pour :

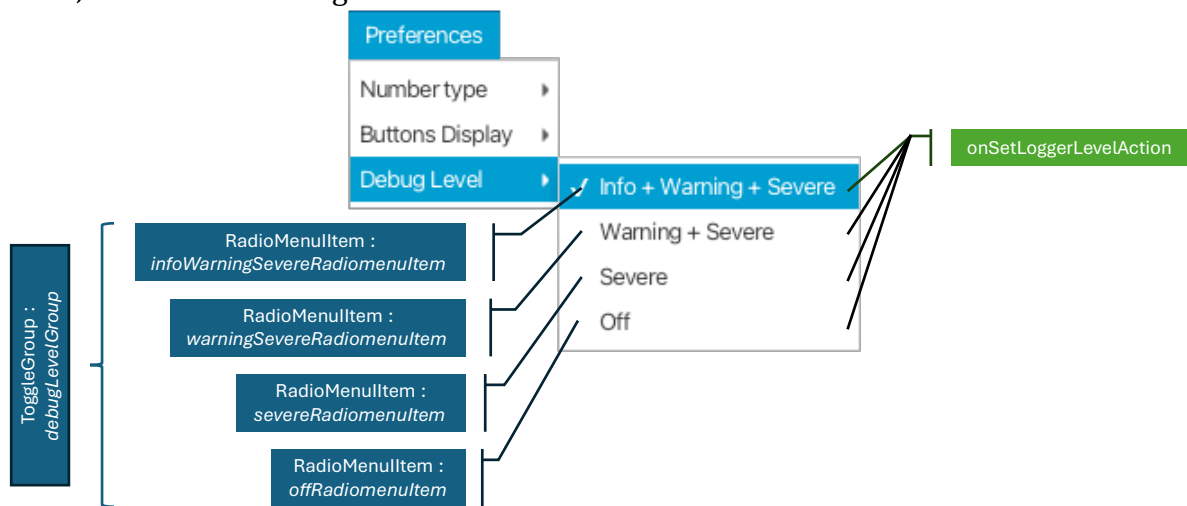
- Être correctement initialisés au démarrage de l'application en fonction des préférences de l'application.
- Pouvoir être sélectionnés programmatically lorsque cette propriété change dans le modèle de données. Ils appartiennent tous au ToggleGroup `numberTypeGroup` afin qu'un seul de ces RadioMenuItem ne soit sélectionné en même temps.



- Pour le menu "Preferences", sous-menu "Buttons display" :
Les 3 RadioMenuItem sont nommés pour être correctement initialisés au démarrage de l'application en fonction des préférences de l'application. Ils appartiennent tous au ToggleGroup `buttonsDisplayGroup`.



- Pour le menu "Preferences", sous-menu "Debug level" :
Les 4 RadioMenuItem sont nommés pour être correctement initialisés au démarrage de l'application en fonction des préférences de l'application. Ils appartiennent tous au ToggleGroup `debugLevelGroup`.



- Pour le menu "About" :



Controller

Le contrôleur comme son nom l'indique contient toute la logique pour interpréter les actions de l'utilisateur dans l'interface graphique et les traduire en opérations sur le modèle de données.

La classe Controller possède un constructeur Controller() dans lequel on pourra initialiser tous les attributs propres du contrôleur (c'est à dire les attributs non annotés @FXML).

Le contrôleur implémente l'interface Initializable, il doit donc implémenter une méthode initialize(URL location, ResourceBundle resources) appelée *après* le constructeur Controller() et dans laquelle nous allons initialiser tous les attributs et propriétés liés à JavaFX :

Vous pourrez remarquer que tous les attributs annotés @FXML sont nulls dans le constructeur Controller() alors qu'ils sont tous non nulls dans la méthode initialize : Ils ont été créés puis injectés par le FXMLLoader du programme principal.

Les attributs propres du contrôleur (c'est à dire les attributs non annotés @FXML) sont les suivants :

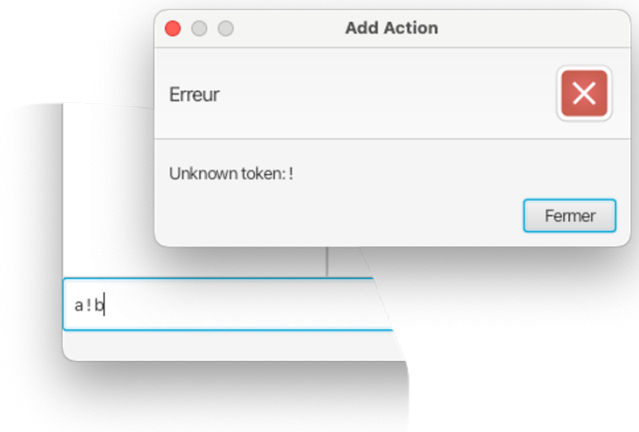
- private Logger logger : un [Logger](#) que nous utiliserons pour écrire (ou pas) des messages sur la console (ou dans un fichier de log). Le Logger dispose de méthodes info(String msg), warning(String msg) et severe(String msg) permettant d'indiquer le niveau de gravité du message.
- private Level loggerLevel : Niveau de log ([Level](#)) minimum des messages à afficher sur la console par le Logger. Par exemple si le niveau Level.WARNING est affecté au Logger celui-ci n'affichera sur la console que les messages avec un niveau de log supérieur ou égal à Level.WARNING : donc Level.WARNING et Level.SEVERE en l'occurrence.
- private ExpressionsModel<Number> expressionsModel : Notre modèle de données qui contient les expressions arithmétiques, le parser d'expressions et d'autres éléments qui sont détaillés [ici](#).
- private ObjectProperty<Number> specimen : Une propriété JavaFX ([ObjectProperty](#)) contenant un nombre spécimen (Integer, Float ou Double) qui sera utilisé comme spécimen pour le parser du modèle de données.
- private Stage parentStage : une référence à la fenêtre ([Stage](#)) de l'application qui pourra par exemple être utilisé comme parent de boîtes de dialogues.
- private Set<Labeled> styleableButtons : Ensemble de boutons (ou plus exactement ensemble de [Labeled](#) qui est une classe mère de [Button](#)) utilisé pour contenir les boutons sur lesquels on appliquera différents styles (Icône, Icône + texte ou bien texte seul).
- private ContentDisplay contentDisplay : Style ([ContentDisplay](#)) à appliquer aux boutons dans styleableButtons.
- private ObservableList<ExpressionDisplay<Number>> expressionsDisplayList : Liste observable d'ExpressionDisplay<Number> qui sera utilisée comme contenu de la table expressionsTableView.
 - ExpressionDisplay<Number> est une classe permettant d'associer le contenu d'une expression avec sa valeur en vue de les afficher ensembles dans une table ([TableView](#)) : à ce titre elle contient juste deux propriétés : Une propriété texte pour le contenu de l'expression et une propriété [Number](#) pour contenir la valeur de l'expression.


N.B. : Tous les éléments dit "Observables" en JavaFX peuvent être utilisés comme contenus de conteneurs JavaFX dans la mesure où ces conteneurs deviennent des "observateurs" d'un contenu "observable". De ce fait, à chaque fois qu'un contenu observable change, le conteneur qui observe ce contenu est notifié du changement et peut donc se redessiner.

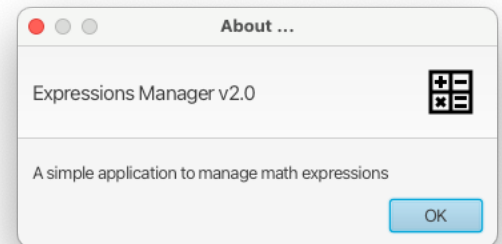
- private ObservableList<VariableDisplay<Number>> variablesDisplayList : Liste observable de VariableDisplay<Number> qui sera utilisée comme contenu de la table variablesTableView.
 - VariableDisplay<Number> est une classe permettant d'associer un nom de variable ainsi qu'une valeur numérique en vue de les afficher ensemble dans une table (tout comme ExpressionDisplay<Number> précédemment. D'ailleurs ces deux classes ont une classe mère commune : NamedDataDisplay<E extends Number>)

Les actions du contrôleur sont les suivantes :

- `public void onQuitAction(ActionEvent event)` : pour quitter l'application. qui elle-même fait appel à la méthode utilitaire `protected void quitActionImpl(Event event)`.
- `public void onAddAction(ActionEvent event)` : Parse la ou les expressions présentes dans `inputField` et les ajoute aux expressions déjà présente si les expressions parsées ne sont pas déjà présentes. Si une erreur de parsing survient, ouvrir une boite de dialogue (Par exemple une [Alert](#)) :
- `public void onEditCommitAction(CellEditEvent<ExpressionDisplay<? extends Number>, String> event)` : Permet d'éditer une cellule du tableau `expressionsTableView` et de re-parser son contenu pour en extraire une ou plusieurs expressions. De la même manière que précédemment, si une erreur de parsing intervient, ouvrir une boite de dialogue pour signaler cette erreur.
- `public void onDeleteSelectedAction(ActionEvent event)` : Permet de supprimer la ou les expressions sélectionnées. Si une variable n'apparaissait que dans les expressions supprimées elle devra disparaître de la liste des noms de variables.
- `public void onSelectAllAction(ActionEvent event)` : Permet de sélectionner toutes les expressions.
- `public void onDeselectAllAction(ActionEvent event)` : Permet de désélectionner toutes les expressions.
- `public void onNewAction(ActionEvent event)` : Permet de repartir de zéro en effaçant toutes les expressions. On considérera alors qu'aucun fichier n'a été encore chargé.
- `public void onOpenAction(ActionEvent event)` : Permet d'effacer toutes les expressions présentes, puis d'ouvrir un fichier texte contenant une ou plusieurs expressions, de les parser et de les ajouter. Si une ligne commence par le mot clé "type" et se termine par "int | float | double" cela permet de mettre en place ce type nombres dans le parser. (Voire aussi l'action `onSelectNumberType`).
- `public void onAppendAction(ActionEvent event)` : Même action que la précédente mais les expressions courantes ne sont pas effacées au préalable. Ce qui suppose que le type de nombre des expressions à lire depuis le fichier reste inchangée par rapport au type actuel afin que toutes les expressions aient le même type de nombres.
- `public void onSaveAction(ActionEvent event)` : Sauve les expressions dans le fichier texte (celui qui a été lu lors des actions `onAppendAction` ou `onAppendAction`). Cette action doit donc rester désactivée tant qu'un fichier n'a pas été lu au préalable.
- `public void onSaveAsAction(ActionEvent event)` : Sauve les expressions dans un fichier texte choisi par l'utilisateur.
- `public void onRevertAction(ActionEvent event)` : Efface les expressions courantes et recharge le fichier lu précédemment. Là aussi cette action doit rester désactivée tant qu'un fichier n'aura pas été lu au préalable.
- `public void onSelectNumberType(ActionEvent event)` : Permet de mettre en place un type de nombre dans le parser (et donc incidemment dans toutes les expressions produites par le parser). Les expressions présentes dans le modèle sont effacées et re-parsées avec le nouveau parser (au risque de produire des erreurs si les expressions ne peuvent pas être parsées par le nouveau parser) : On pourra pour se faire sauvegarder toutes les expressions dans une chaîne puis re-parser cette chaîne.
- `public void onFilterAction(ActionEvent event)` : Permet de modifier les critères de filtrage des expressions dans la table `expressionsTableView` en filtrant soit



- Un contenu texte (par exemple "a" pour conserver toutes les expressions contenant cette lettre).
- Un type d'expression terminale : toutes, variables ou constantes
- Un type d'expression binaire : toutes, affectation, addition, etc.
- `public void onClearFilterAction(ActionEvent event)` : Permet d'effacer tous les critères de filtrage pour revenir à une table `expressionsTableView` affichant toutes les expressions du modèle.
- `public void onDisplayButtonsWithStyle(ActionEvent event)` : Permet de changer le style des boutons de la barre d'outils en haut de l'application pour afficher les boutons :
 - Avec une icône et du texte : 
 - Avec une icône seule : 
 - Avec du texte seul : 
- `public void onSetLoggerLevelAction(ActionEvent event)` : Permet de changer le niveau de debug (`Level loggerLevel`) appliqué aux messages du `Logger logger` apparaissant dans la console.
 - `Level.INFO` : Tous les messages (`logger.info(...)`, `logger.warning(...)`, `logger.severe(...)`) sont affichés sur la console.
 - `Level.WARNING` : Seuls les messages `logger.warning(...)`, `logger.severe(...)` sont affichés sur la console.
 - `Level.SEVERE` : Seuls les messages `logger.severe(...)` sont affichés sur la console.
 - `Level.OFF` : aucun message du logger n'est affiché sur la console.
- `public void onAboutAction(ActionEvent event)` : Ouvre une simple boîte de dialogue présentant l'application :



ExpressionsModel

Cette classe représente notre modèle de données. A ce titre elle contient donc une liste d'Expressions arithmétiques ainsi que d'autres attributs permettant de gérer cette liste d'expressions. Les attributs de notre modèle de données sont donc :

- `private Logger logger` : Un `Logger` pour afficher des messages.
- `private ObjectProperty<E> specimen` : Une propriété contenant un nombre specimen (`Integer`, `Float` ou `Double`) qui sera utilisé pour déterminer le type des nombres dans les expressions.
- `private GroupExpression<E> rootExpression` : une expression (virtuelle puisqu'elle ne réalise aucune opération) pour contenir toutes les expressions de notre modèle.
- `private ObjectProperty<TreeItem<Expression<E>>> rootItem` : une propriété contenant un item d'arbre (`TreeItem`) contenant une `Expression<E>` (dans ce cas la `rootExpression` décrite précédemment). Cette propriété sera utilisée pour représenter la racine de l'arbre des expressions dans un `TreeView`.
- `private ObservableList<Expression<E>> expressions` : La liste des expressions arithmétiques du modèle.
- `private ObservableMap<String, Optional<? extends Number>> variablesMap` : Le dictionnaire reliant les noms de variables (`String`) à leur valeur optionnelle `Optional<? extends Number>` que l'on pourra créer directement à partir du dictionnaire des variables `VariableExpression<E>.getValues()`. Créant ainsi un dictionnaire "observable" que l'on pourra observer dans l'UI. D'ailleurs on peut même spécifier ce dictionnaire comme étant aussi celui des `VariableExpressions` : `VariableExpression.setValues(variablesMap);`, ainsi toute manipulation des noms de variables dans `VariableExpression` sera en même temps faite dans `ExpressionsModel`.

- `private Predicate<Expression<E>> predicate` : Prédicat que l'on va utiliser pour "filtrer" la liste des expressions `expressions` en une liste filtrée `filteredExpressions`. Ce prédicat devra être réactualisé à chaque fois que l'une des propriétés `operatorFiltering`, `operandFiltering`, `nameFiltering` changera : On pourra pour ce faire ajouter des "listeners" à ces propriétés pour qu'à chaque modification le `predicate` soit recréé en conséquence.
- `private FilteredList<Expression<E>> filteredExpressions` : Liste d'expressions résultant du filtrage de expressions par `predicate`
- `private ObjectProperty<BinaryOperatorRules> operatorFiltering` : Propriété participant au filtrage des expressions en se basant sur le type de `BinaryExpression<E>` : addition, soustraction, multiplication, division, puissance, ou bien n'importe quelle expression binaire. Par exemple si la propriété vaut `BinaryOperatorRules.ASSIGNMENT` on ne conservera dans les expressions filtrées que celle contenant une affectation `=`. Et si la propriété vaut `BinaryOperatorRules.ANY` on conservera toutes les expressions binaires.
- `private ObjectProperty<TerminalType> operandFiltering` : Propriété participant au filtrage des expressions en se basant sur le type d'expression terminales que doivent contenir les expressions à conserver lors du filtrage. Par exemple si cette propriété vaut `TerminalType.CONSTANTS` on ne conservera dans les expressions filtrées que celles contenant des constantes.
- `private StringProperty nameFiltering` : Propriété participant au filtrage des expressions en se basant sur du texte. On ne conservera dans les expressions filtrées que celle dont la représentation sous forme de chaîne de caractères (celle obtenue avec `toString()`) contient le contenu de cette propriété.
- `private ExpressionParser<E> parser` : Le parser à utiliser pour parser un contexte et ainsi obtenir de nouvelles expressions. Il faudra recréer ce parser à chaque fois que le type de nombre dans les expressions changera.
- `private File file` : Instance du fichier utilisé pour lire les expressions depuis un fichier ou bien pour écrire les expressions vers un fichier. Tant qu'un fichier n'aura pas été lu cette instance peut être null.
- `private ReadOnlyBooleanWrapper hasFile` : propriété booléenne indiquant qu'un fichier a été lu (et donc que `file` est non null). `hasFile` n'est pas directement une propriété mais un "Wrapper" qui permet de changer sa valeur (avec la méthode `set`) mais qui sera fournie à l'extérieur sous la forme `ReadOnlyBooleanProperty` afin qu'elle ne puisse pas être modifiée.

Vous pouvez constater que cette classe contient de nombreuses propriétés JavaFX au lieu de simples attributs. La raison est la suivante : Les propriétés JavaFX (Voire l'interface [Property<E>](#)) peuvent être partagées. Une propriété peut "observer" une autre propriété, auquel cas tout changement dans la propriété "observée" sera automatiquement reporté dans la propriété "observatrice". Exemple:

```
Property<Number> aProperty = myModel.specimenProperty();
Property<Number> bProperty = specimenProperty();
bProperty.bind(aProperty);
    // Tout changement dans aProperty sera reporté dans bProperty
boolean isBound = bProperty.isBound();
if (!isBound)
{
    bProperty.set(Integer.of(0)); // set : Illegal si isBound est vrai
}
bProperty.unbind();
bProperty.bindBidirectional(aProperty);
    // Tout changement dans aProperty sera reporté dans bProperty
    // Et tout changement dans bProperty sera reporté dans aProperty
// Ajout d'un Listener à aProperty qui sera appelé à chaque chgt de aProperty
aProperty.addListener((observable, oldValue, newValue) -> {
    logger.info(observable + " changed from " + oldValue + " to " + newValue);
});
```

Hors les éléments d'interface graphique de JavaFX possèdent eux aussi de nombreuses propriétés. Nous allons donc profiter de ces mécanismes pour connecter des propriétés de l'UI aux propriétés de notre modèle ou bien l'inverse. En voici quelques exemples :

- `TreeView.rootProperty()` → `ExpressionsModel.rootItemProperty()` : pour connecter le contenu d'un `TreeView` avec l'élément racine de nos expressions dans le modèle.
- `ExpressionsModel.operatorFilteringProperty()` → `ComboBox.valueProperty()` : pour connecter la propriété de filtrage des expressions en fonction du type d'opération de notre modèle à la valeur sélectionnée dans un `ComboBox` (liste déroulante).
- `ExpressionsModel.nameFilteringProperty()` → `TextField.textProperty()` : pour connecter la propriété de filtrage des expressions en fonction du contenu de notre modèle au contenu d'un `TextField` dans l'UI.
- `Button.disabledProperty()` → `ExpressionsModel.hasFileProperty().not()` : pour rendre un `Button` de l'UI désactivé si la propriété booléenne de notre modèle indiquant qu'un fichier a été ouvert est fausse. On aura inversé la propriété du modèle grâce à la méthode `not()` de [BooleanExpression](#).

Une grande partie de l'initialisation du `Controller` consistera donc à connecter des propriétés de notre modèle à des propriétés de l'UI ou vice-versa.

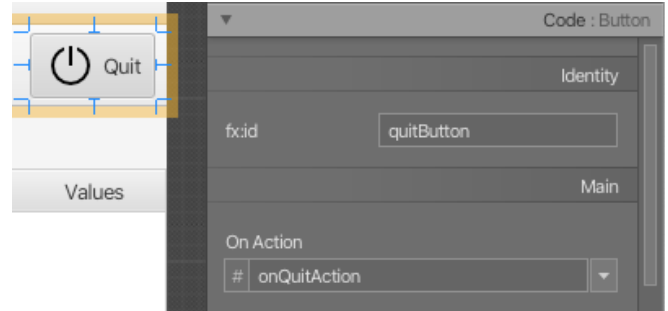
Les opérations réalisées par le modèle de données sont les suivantes

- `public ObservableList<Expression<E>> getExpressions()` : permet d'obtenir la liste (filtrée) des expressions.
- `public ObservableMap<String, Optional<? extends Number>> getVariables()` : permet d'obtenir le dictionnaire des variables.
- `public File getFile()` : permet d'obtenir le fichier dans lequel lire ou écrire.
- `public void resetFile()` : permet d'oublier le fichier. `file` est remis à `null` et `hasFile` est mis à `false`;
- `public void setNumberType(Number specimen)` : permet de changer le parser pour qu'il puisse parser des expressions contenant des nombres correspondant au specimen fourni.
- `public void clear()` : permet d'effacer toutes les données du modèle
 - Les expressions.
 - Les variables : `variablesMap`.
 - L'expression parente de toutes les expressions : `rootExpression`.
 - l'item d'arbre contenant cette expression parente : `rootItem`.
- `public boolean parse(String context)` : permet d'ajouter au modèle les expressions parsées depuis le contexte.
- `public boolean reparse(Expression<? extends Number> expression, String context)` : Permet de remplacer dans le modèle l'expression passée en premier argument (si elle existe) par le parsing du contexte passé en second argument. Cette opération intervient lorsque l'on édite une expression par exemple. Si le contexte est `null` ou vide on considérera que l'on doit retirer l'expression en premier argument du modèle (voire l'opération suivante).
- `public boolean remove(Expression<? extends Number> expression)` : Permet de retirer du modèle l'expression passée en argument (si elle existe).
- `public boolean load(File file, boolean append)` : Permet de lire le contenu du fichier texte passé en argument. Cela consiste à parser chaque ligne de ce fichier comme un nouveau contexte et ajouter la ou les expressions résultantes aux expressions du modèle. Lorsque le second argument `append` est faux le modèle courant est effacé avant d'être remplacé par les expressions lues depuis le fichier.
 - Les différentes expressions dans le fichier peuvent être séparées par un saut de ligne ou bien un `","`

- Si une ligne du fichier commence par le mot "type" elle doit se terminer par "int", "float" ou "double" pour indiquer le type de nombres que contiennent les expressions dans ce fichier afin que l'on puisse mettre en place un parser adéquat.
- `public boolean save(File file)` : Permet d'écrire les expressions du modèle dans un fichier texte. On pourra pour se faire compter sur la méthode `toString()` de chaque type d'expression.
 - Avec une particularité : S'il existe dans le dictionnaire des variables, des variables avec une valeur qui ne sont pas représentées par une expression, alors on créera des opérations binaires d'affectation pour représenter des variables. Par exemple si l'on a une variable *a* avec une valeur 2 dans le dictionnaire des variables alors qu'aucune des expressions du modèle ne fournit de valeur à la variable *a* on créera (au moment de l'écriture vers le fichier et sans ajouter cette expression au modèle) une affectation *a* = 2 que l'on écrira dans le fichier.
- `public String toString()` : Permet de créer une chaîne de caractère contenant toutes les expressions du modèle séparées par des ";".
 - Comme précédemment, s'il existe dans le dictionnaire des variables, des variables avec une valeur qui ne sont pas représentées par une expression, alors on créera des opérations binaires d'affectation pour représenter ces variables dans la chaîne résultante.
- Chaque propriété du modèle engendre 3 méthodes pour y accéder (ce que l'on appelle les accesseurs JavaFX). Par exemple pour la propriété private `ObjectProperty<E> specimen` on aura donc 3 méthodes :
 - `public final ObjectProperty<E> specimenProperty()` pour obtenir la propriété elle-même.
 - `public final E getSpecimen()` pour obtenir la valeur de la propriété
 - `public final void setSpecimen(final E value)` pour mettre en place une nouvelle valeur dans la propriété si et seulement si celle-ci n'est pas liée à une autre propriété (`!specimen.isBound()`) sans quoi une erreur d'exécution interviendra.
- `public void refreshRoot()` : Pour rafraîchir le nœud parent de l'arbre des expressions afin que celui-ci se mette à jour dans le `TreeView`.
- Le reste des méthodes sont des méthodes utilitaires privées utilisées dans les opérations décrites ci-dessus :
 - `private boolean merge(List<Expression<E>> expressions)` : permet d'ajouter les expressions contenues dans la liste en argument aux expressions du modèle sans doublons. On veillera donc à ne pas ajouter au modèle une expression qui y est déjà présente.
 - `private AssignmentExpression<E> getAssignmentFor(String name)` : Permet de trouver parmi les expressions du modèle une expression binaire d'affectation donnant une valeur à la variable dont le nom est passé en argument. Si le résultat de cette méthode est null cela indique qu'aucune expression du modèle ne donne une valeur à la variable dont le nom est passé en argument.
 - `private void setPredicate(BinaryOperatorRules operatorFilter, TerminalType operandFilter, String searchName)` : Permet de mettre en place dans l'attribut `predicate` un nouveau prédicat filtrant les expressions du modèle à partir des valeurs passées en argument.
 - `private static <E extends Number> boolean searchFor(Expression<E> expression, TerminalType type)` : Permet de chercher dans l'expression la présence d'une expression terminale d'un certain type (constante, variable, ou bien l'une des deux).
 - `private static <E extends Number> boolean containsVariable(Expression<E> expression, String name)` : Permet de chercher la présence d'une `VariableExpression` nommée *name* dans l'expression *expression*.
 - `private boolean cleanupVariablesMap()` : Permet de nettoyer le dictionnaire des variables. Retire du dictionnaire des variables toutes les entrées qui ne sont pas présentes dans au moins une des expressions du modèle.

Travail à réaliser

1. Importez vos Expressions<E> dans le projet. Vous pourrez vérifier leur bon fonctionnement avec les classes de test JUnit 5 du package tests.
2. Complétez le fichier FXML ExpressionsFrame.fxml avec le logiciel SceneBuilder (ou même en éditant directement le fichier) pour que les différents éléments de l'UI appellent les bons callbacks dans le Controller. Exemple : dans SceneBuilder ce qui correspond dans le fichier ExpressionsFrame.fxml à :



```
<Button fx:id="quitButton"
mnemonicParsing="false" onAction="#onQuitAction" text="Quit">
...
</Button>
```

- Chaque attribut annoté @FXML dans la classe Controller doit se retrouver dans le fichier FXML.
 - Tous les callbacks onXXXAction(...) du Controller doivent être référencés dans le fichier FXML.
3. Complétez la classe expressions.models.ExpressionsModel. Vous pourrez la tester avec la classe de test JUnit tests.TestExpressionsModel
 4. Complétez la classe Controller :
 - Son constructeur Controller() pour initialiser ses attributs non JavaFX.
 - Sa méthode initialize(URL location, ResourceBundle resources) pour initialiser l'UI, créer les connections entre propriétés ou mettre en place des listeners.
 - Les différents callbacks onXXXAction(...) pour réaliser les différentes opérations de notre application.

Vous aurez jusqu'au 16/05/2025 pour déposer une archive de votre projet sur exam.ensiie.fr sur le dépôt laob-tp4.

TODOs

La plupart des éléments de code à compléter sont indiqués dans le code par des tags TODO. Vous pourrez aussi trouver des tags DONE : Dans ce cas considérez le code muni d'une telle annotation comme un exemple réalisé pour d'autres TODOs.

TODO Tree dans VSCode

l'expression régulière pour chercher les "TODOs" dans VSCode ne permet pas de trouver les TODOs dans les commentaires multi-lignes tels que

```
/*
 * TODO Do stuff
 */
```

C'est pourquoi il est recommandé de la changer les settings de Todo-tree > Regex : Regex de (//|#|<!--|;|/*|^\^[\t]*(-|\d+.)\s*(\$TAGS) vers (//|#|<!--|;|/*|^\^[\t]*(-|\d+.)\s*(\$TAGS) pour voir apparaître tous les TODOs.

Annexes

Installation et utilisation de JavaFX

D'après le tutoriel [Getting Started with JavaFX](#)

Depuis Java 11, JavaFX ne fait plus partie de la distribution standard de Java afin de réduire la taille du SDK. Il faut donc installer JavaFX séparément.

Installation

À l'école la version 20 de JavaFX pour Linux est installée dans `/pub/FISE_LA0B12/javafx-sdk-20/lib`. Néanmoins, si vous souhaitez installer JavaFX sur votre machine, vous pouvez le télécharger à l'adresse <https://gluonhq.com/products/javafx/>. Veillez à bien choisir l'archive pour votre système d'exploitation (*Linux*, *MacOS* ou *Windows*), votre architecture (a priori *x64* si vous avez un processeur Intel et *aarch64* autrement) ainsi que le type d'archive, ici *SDK*, ainsi que la version de JavaFX adaptée à votre version de Java (Par exemple la version 24 de JavaFX requiert la version 22 de Java).

Configuration

Une fois JavaFX installé, il faut préciser à Eclipse où se trouve JavaFX en

- créant une nouvelle bibliothèque utilisateur ("User Library") : Preferences -> Java -> Build Path -> User Libraries -> New
- que vous pouvez appeler "JavaFX" (vous pouvez la considérer comme une System Library)
- Vous allez ensuite ajouter à cette bibliothèque les "external JARs" qui se trouvent à l'endroit où JavaFX est installé. En l'occurrence ici : `/pub/FISE_LA0B12/javafx-sdk-20/lib`
- Vous pourrez alors ajouter cette bibliothèque à votre projet JavaFX : Clic droit à la racine de votre projet : Properties -> Java Build Path -> Onglet Libraries -> Add Library (à classpath) -> User Library -> Next -> choisissez la bibliothèque JavaFX que vous avez créée à l'étape précédente.

Lancement d'un programme JavaFX

Pour pouvoir lancer un programme principal Java utilisant JavaFX (voir la classe `Main` dans le package `application`), Il ne suffit **PLUS** d'un clic droit sur la classe `Main` -> Run As ... -> Java Application : vous obtiendrez une erreur car le runtime java ne sais pas où se trouve JavaFX. Il faut donc créer une **Run Configuration** : clic droit sur la classe `Main` -> Run As ... -> Run Configurations ...

- Double cliquez sur "Java Application" dans la colonne de gauche pour créer une nouvelle configuration que vous pourrez appeler "Converter" puis dans l'onglet "Arguments" de la partie droite :
 - Ajoutez les "VM arguments" : `--module-path /pub/FISE_LA0B12/javafx-sdk-20/lib`
`--add-modules javafx.controls,javafx.fxml` (sur une seule ligne).
 - A terme, vous pourrez remplacer `/pub/...` par une variable `${PATH_TO_FX}` pointant vers `/pub/FISE_LA0B12/javafx-sdk-20/lib`
 - **Désélectionnez** la case "Use the -XStartOnFirstThread argument when launching with SWT" si elle est présente.

Les instructions mentionnées ici sont tirées des tutoriels originaux de JavaFX pour tous les IDEs :

1. [IntelliJ](#)
2. [NetBeans](#)
3. [Eclipse](#)
4. [Visual Studio Code](#)
5. Suivez les instructions pour les *Non-modular projects*.

Installation de SceneBuilder

A l'école SceneBuilder est installé dans /opt/scenebuilder. Vous pourrez donc le lancer avec la commande :
/opt/scenebuilder/bin/SceneBuilder

Si vous souhaitez installer SceneBuilder sur votre machine, celui-ci est téléchargeable à l'adresse <https://gluonhq.com/products/scene-builder/>. Il n'a (a priori) pas besoin de JavaFX car il contient son propre runtime Java + JavaFX.

Plugins JavaFX pour votre IDE

Si vous utilisez Eclipse vous pourrez installer le plugin **e(fx)clipse** ([update site](#)) pour utiliser SceneBuilder directement depuis eclipse. Suivez cette [procédure](#).

Sur les machines de l'école vous pourrez lancer une version d'eclipse équipée de [e(fx)clipse] en lançant :
/pub/FISE_LAOB12/efxclipse/eclipse au lieu de taper simplement eclipse. Cela ne vous dispensera toutefois pas d'importer la librairie JavaFX dans votre projet.

Vous pouvez tout à fait utiliser Scenebuilder sans connexion avec Eclipse, néanmoins, chaque fois que vous éditez un fichier FXML avec Scenebuilder il faudra **impérativement** rafraîchir votre projet Eclipse (F5) avant de lancer l'application, sans quoi le fichier FXML mis à jour ne sera pas pris en compte.

Lecture et Écriture d'un fichier Texte en Java

La façon la plus simple de lire ou d'écrire dans des fichiers texte en Java consiste à utiliser ce que l'on appelle des try(with ressources){...} dans lesquels les ressources sont automatiquement fermées après l'accolade fermante.

Lecture d'un fichier texte :

```
File file = ...
...
try (FileReader fr = new FileReader(file))
{
    try (BufferedReader br = new
BufferedReader(fr))
    {
        String line = null;
        do
        {
            line = br.readLine();
            if (line != null)
            {
                // do stuff with line
            }
        } while (line != null);
    }
}
```

Écriture d'un fichier texte :

```
File file = ...
Stuff stuff = ...
...
try (FileWriter writer = new
FileWriter(file))
{
    try (PrintWriter printer = new
PrintWriter(writer))
    {
        printer.println("printing " + stuff
+ " to file");
    }
}
```