

Introduction à la programmation des interfaces graphiques (JavaFX)

Table des matières

I.	Généralité.....	2
II.	Introduction à javaFX	3
1.	Architecture d'une interface en JavaFX	4
2.	Les composants graphiques de JavaFX.....	5
3.	Les classes JAVA FX.....	6
4.	JAVA FX FXML Application.....	6
5.	Scene Builder	10
III.	Première application JavaFX	13
1.	Création de l'interface graphique	13
2.	Affichage de l'interface graphique via la classe principale	19
3.	Gestion des évènements : lier un composant à une action	19
2.1	Renseigner les éléments dans la combobox.....	19
2.2	Créer des actions pour les boutons enregistrer et fermer.....	20
2.3	Action d'enregistrement :	20
3.1	Action de fermeture :	22
4.	Gestion des listes en JAVA FX : TableView	24
4.1	Utilisation des Tables pour afficher les clients par colonne :.....	24
4.2	Modification et suppression d'un client en JAVA FX à partir de la table	26

I. Généralité

Les interfaces graphiques (ou interfaces homme-machine) sont appelées GUI (pour Graphical User Interface). Elles permettent à l'utilisateur d'interagir avec un programme informatique, grâce à différents objets graphiques (boutons, menus, cases à cocher...). Ces objets sont généralement actionnés à l'aide de la souris ou du clavier.

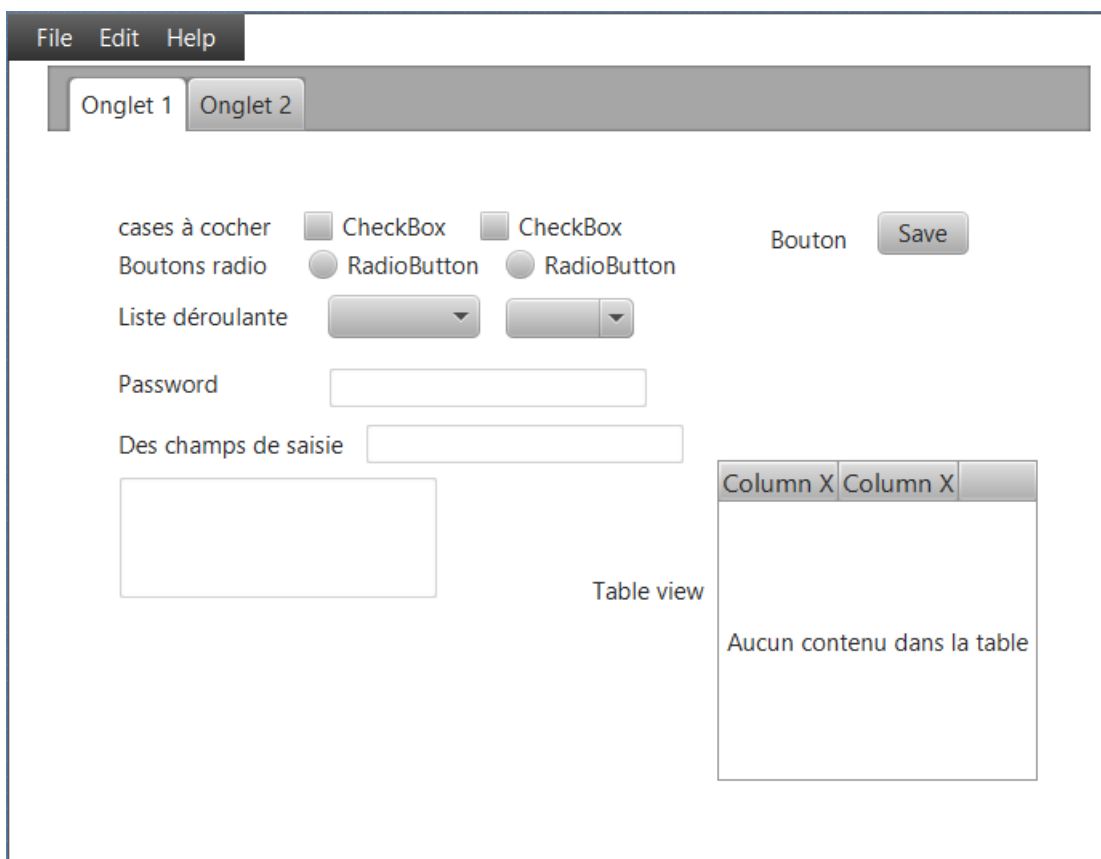
Malgré le fait que les interfaces graphiques semblent secondaires par rapport au développement du cœur d'une application, elles doivent néanmoins être conçues et développées avec soin et rigueur.

Leur efficacité et leur ergonomie sont essentielles dans l'acceptation et l'utilisation de ces outils par les utilisateurs finaux. Une bonne conception et un développement maîtrisé permettent également d'en assurer une meilleure maintenabilité.

Rôles d'une interface utilisateur : (1) Montrer – (2) Réagir

- montrer le résultat de l'exécution
- permettre à l'utilisateur d'interagir

Vous avez forcément remarqué les interfaces graphiques suivantes :



La bibliothèque Java contient trois ensembles de classes permettant la création d'interfaces utilisateurs graphiques (graphical user interfaces ou GUIs). Dans l'ordre chronologique de leur apparition, il s'agit de :

- AWT (Abstract Window Toolkit), tombé en désuétude mais servant de base à Swing,
- Swing, utilisé pour la plupart des applications jusqu'à récemment, mais en cours de remplacement par JavaFX, et
- JavaFX : Java **Flash** + **flex** : qui permet de gérer en plus des interfaces graphiques basiques, des animations et des effets.

Ce tutorial présente une vue d'ensemble des principaux concepts de JavaFX, la plupart d'entre eux se retrouvant, sous une forme ou une autre, dans des bibliothèques similaires pour d'autres langages de programmation. Au vu de la taille de la bibliothèque JavaFX, il est impossible ici de la décrire de manière détaillée.

II. Introduction à javaFX

JavaFX est une technologie créée par Sun Microsystems qui appartient désormais à Oracle. Avec l'apparition de Java 8 en mars 2014, JavaFX devient la bibliothèque de création d'interface graphique officielle du langage Java, pour toutes les sortes d'application (applications mobiles, applications sur poste de travail, applications Web).

JavaFX est le successeur officiel de Swing. La version **JavaFX 8**, sortie avec [Java 8](#), permet de développer des clients riches tout en simplifiant leur développement grâce au langage **FXML** et l'outil **SceneBuilder** d'Oracle.

JavaFX permet au développeur de créer, dessiner, tester, déboguer et déployer des applications client riche qui fonctionnent de façon cohérente sur des plateformes différentes.

La richesse de l'API autorise des effets visuels comme la manipulation de contenu multimédia. JavaFX contient des outils très divers, notamment pour les médias audio et vidéo, le graphisme 2D et 3D, la programmation Web, etc.

NB : Le SDK (Software Development Kit) de JavaFX étant désormais intégré au JDK standard Java SE.

Java FX propose :

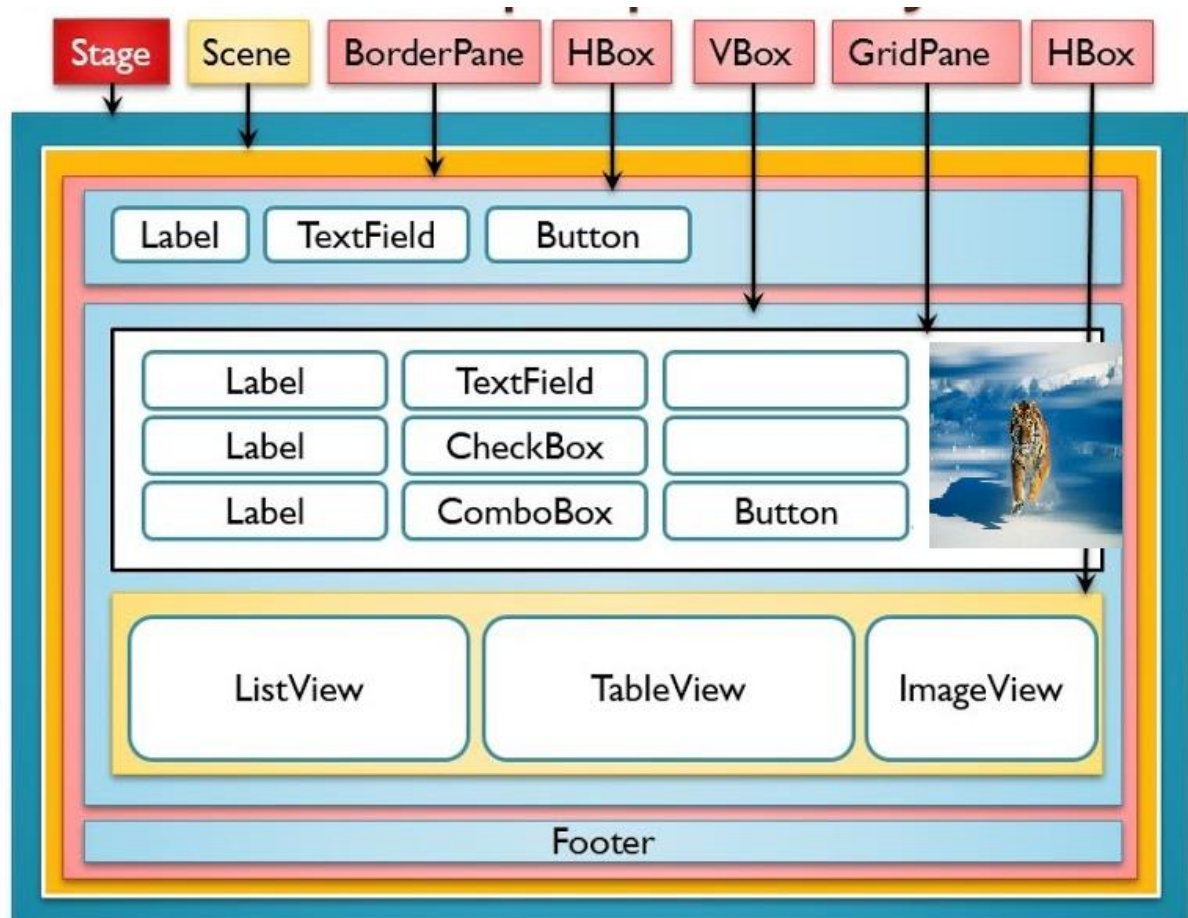
- une API (des classes java) pour la gestion de graphiques, fenêtres
- un logiciel de création de fenêtres de dialogue 'Scene Builder', générant des fichiers FXML
- un composant de lecture de page web (WebView)

- un support 'MultiTouch' pour les applications mobile
- des composants pour la conception d'images 3D
- des liens vers des composants 'Swing' (ancien mode de création d'interfaces graphiques)

1. Architecture d'une interface en JavaFX

Une interface graphique JavaFX est constituée d'un certain nombre de nœuds (nodes), qui peuvent être de différente nature : simples formes géométriques (cercles, lignes, etc.), composants d'interface utilisateur (bouton, menu, etc.), conteneurs, etc. Ces nœuds sont organisés en une hiérarchie que l'on nomme le graphe de scène (scene graph). Malgré son nom, le graphe de scène est un simple arbre, c'est-à-dire un graphe acyclique dans lequel chaque nœud a au plus un parent. Cet arbre reflète l'organisation des nœuds à l'écran, dans la mesure où un descendant d'un nœud dans le graphe de scène apparaît généralement à l'écran imbriqué dans son ancêtre.

Ce schéma représente la structure d'une application JavaFX :



Une application JavaFX est constituée de :

1. Le premier élément dont elle est constituée est un objet Stage, c'est dans cet objet que tout se passe, il représente la fenêtre de notre application.
2. A l'intérieur de cet objet Stage il y a un objet Scene, Tout ce qui apparaîtra dans notre application devra y être inséré.
3. Enfin, l'objet Scene contient des nœuds graphiques. Ces nœuds graphiques sont des objets qui peuvent être de différents types : des boutons, des zones textes, des libellés, des cercles, des rectangles, des images...

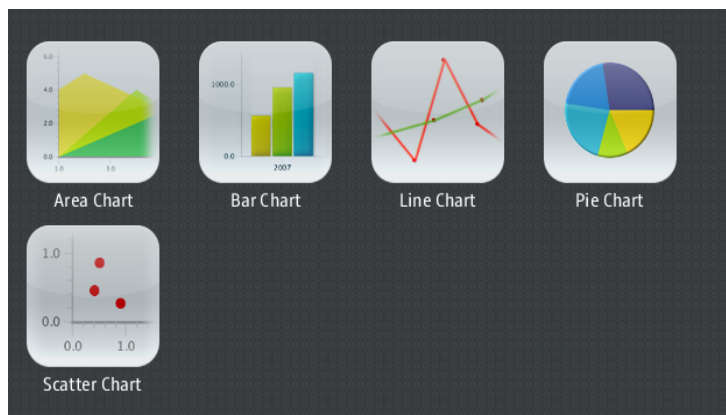
2. Les composants graphiques de JavaFX

Les éléments de contrôle utilisateur sont tous les éléments qui permettent à l'utilisateur d'indiquer des informations le concernant, vous les connaissez forcément : les boutons, les champs texte, mot de passe, les sliders, les cases à cocher...

La librairie JavaFX offre un ensemble de composants (kit de développement) pour créer les interfaces utilisateurs graphiques.



JAVAFX permet aussi d'intégrer des composants graphiques gérant des graphes tels que :



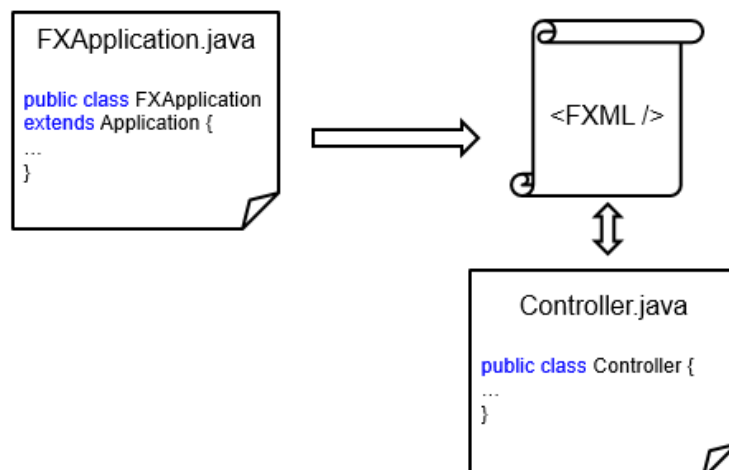
3. Les classes JAVA FX

JavaFX Main Class	Extends JavaFX 'Application' Contains main() and launch() launch() will initialise and call start() load the fxml file containing the GUI
FXML	XML based language Describes UI appearance Controlled by Scene Builder
Stage	Represents the main container and window
Scene	Contains All graphic components
FXMLLoader	Handles parsing of FXML files into Java Class @FXML keyword links private Java variables to corresponding FXML components @FXML private TableView<Person> personTable;
JavaFX Collections	Data Structures that have Event Listeners Can handle changes made in GUI automatically ObservableList is example of array with Listener

4. JAVAFX FXML Application

JAVAFX utilise des fichiers FXML, basés sur le langage XML, pour définir les interfaces utilisateur.

L'architecture d'une application JAVAFX en se basant sur FXML est défini comme suit :



A) Le fichier FXML

Le fichier FXML est un fichier au format XML dont la syntaxe est conçue pour décrire l'interface (la vue) avec ses composants, ses conteneurs, sa disposition, ...

- Le fichier FXML décrit le "quoi" mais pas le "comment"

Ces fichiers XML vont permettre de décrire notre interface de manière un peu plus visuelle que le code brut. De plus, il est possible de binder les éléments FXML directement sur des classes Java. C'est ce que nous allons faire pour créer notre interface.

A l'exécution, le fichier FXML sera chargé par l'application (classe FXMLLoader) et un objet Java sera créé (généralement la racine est un conteneur) avec les éléments que le fichier décrit (les composants, conteneurs, graphiques, ...).

Le langage FXML n'est pas associé à un schéma XML mais la structure de sa syntaxe correspond à celle des API JavaFX :

- Les classes JavaFX (conteneurs, composants) peuvent être utilisées comme éléments dans la syntaxe XML
- L'attribut `fx:id` permet de définir un identifiant de chaque composant.
- Les propriétés des composants correspondent à leurs attributs.

B) Contrôleur

Il est nécessaire de créer une classe qui permettra d'effectuer des actions suite aux actions de l'utilisateur. Il s'agit d'une classe 'normale' qui contient des attributs et méthodes partagés avec la fenêtre FXML. Ces attributs et méthodes doivent être précédés du marqueur **@FXML**.

Cette classe séparée joue le rôle de contrôleur pour traiter l'action du clic sur le bouton ou autre évènement. Le principe de fonctionnement et le rôle des annotations seront expliqués dans les pages qui suivent.

```
public class SayHelloController {

    @FXML
    private Button btnHello; // Object injected by FXMLLoader (fx:id="btnHello")

    @FXML
    private Label title; // Object injected by FXMLLoader (fx:id="title")

    @FXML
    private void handleButtonAction(ActionEvent event) {
        title.setText("H e l l o  !");
        title.setTextFill(Color.FUCHSIA);
    }
}
```

La classe qui joue le rôle de contrôleur pour une interface déclarée en FXML doit être annoncée dans l'élément racine, en utilisant l'attribut `fx:controller` :

```
<BorderPane prefHeight="80.0" prefWidth="250.0"
    style="-fx-background-color: #FFFCAA;"
    xmlns=http://javafx.com/javafx/8
    xmlns:fx=http://javafx.com/fxml/1
    fx:controller="supp_cours.chap07.SayHelloController">
    . . .
```

C) Liens FXML ↔ programme (Contrôleur)

Le lien entre les composants décrits dans le fichier FXML et le programme est établi par les attributs `fx:id` :

```
<Label id="title" fx:id="title" text="Titre" textFill="#0022cc" ...
```

L'attribut `fx:id` fonctionne en lien avec l'annotation `@FXML` que l'on peut utiliser dans les contrôleurs, et qui va indiquer au système que le composant avec le nom `fx:id` pourra être injecté dans l'objet correspondant de la classe contrôleur.


```
public class SayHelloController {
    @FXML
    private Button btnHello;

    @FXML
    private Label title;    // fx:id="title"
                          // Object injected by FXMLLoader
}
```

Pour les composants actifs déclarés dans une interface en FXML, on peut indiquer la méthode du contrôleur qui doit être invoquée en utilisant l'attribut `onAction="#methodName"` :

```
<Button fx:id="btnHello" onAction="#handleButtonAction"
        text="Say Hello" BorderPane.alignment="CENTER" />

. . .
```

Un programme doté d'une interface graphique ne fait généralement qu'attendre que l'utilisateur interagisse avec celle-ci, réagit en conséquence, puis se remet à attendre. Chaque fois que le programme est forcé de réagir, on dit qu'un événement (event) s'est produit. Cette caractéristique des programmes graphiques induit un style de programmation particulier nommé programmation événementielle (event-driven programming).

Dans la classe contrôleur, ces méthodes devront (comme les composants associés) être annotées avec `@FXML`.

```
@FXML
private void handleButtonAction(ActionEvent event) {
    title.setText("H e l l o   !");
    title.setTextFill(Color.FUCHSIA);
}
```

D) Liens FXML ↔ programme (La Classe FxApplication.java)

La classe Main contient deux fonctions :

- La fonction `main()`. c'est le point d'entrée de notre application. Elle appelle la fonction `launch()` qui lancera le reste du programme. C'est la seule instruction que doit contenir la fonction `main()`.

- La fonction `start()`. Cette fonction est déclenchée par la fonction `launch()`, elle prend en argument un objet de type `Stage`. Vous vous souvenez... c'est le théâtre qui contiendra tout ce qui constitue l'application : la scène, les acteurs, etc... Pour l'instant la fonction `start()` ne fait que rendre visible l'objet `Stage`, c'est-à-dire la fenêtre de notre application.

Elle définit le cycle de vie d'une application JavaFX :

- Au Lancement De L'application, La Méthode `Launch()` Doit Etre Appelée ;
- Cette Méthode Appelle La Méthode `Init()` ;
- Puis La Méthode `Start()` Obligatoirement Implémentée Par Le Client ;
- Une Boucle Est Lancée Pour Gérer Les Evénements ;
- Si Un Signal D'exit Est Lancé, La Méthode `Stop()` Est Appelée.

La méthode `start()` de la classe principale peut charger le fichier FXML en invoquant la méthode statique `FXMLLoader.load(url)` qui prend en paramètre l'URL de la ressource à charger.

La méthode `getResource(name)` de la classe `Class` permet de trouver (par le classloader) l'URL d'une ressource à partir de son nom.

- Référence relative au package courant par défaut ("views/Login.fxml")
- Référence absolue si '/' initial dans le nom ("/resources/log/Error.fxml") (attention: le caractère '/' est aussi utilisé même s'il s'agit de packages et sous-packages).

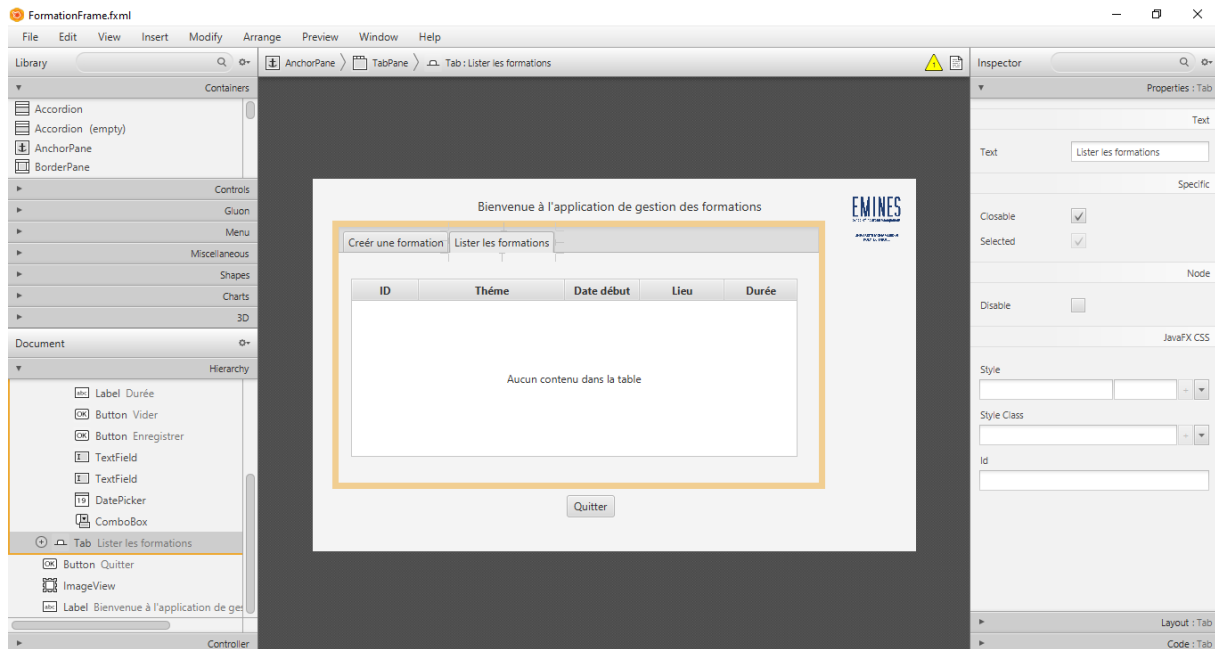
```
public void start(Stage primaryStage) throws Exception {  
    //--- Chargement du fichier FXML  
    BorderPane root = FXMLLoader.load(getClass().getResource("SayHello.fxml"));  
  
    Scene scene = new Scene(root);  
    primaryStage.setScene(scene);  
    primaryStage.setTitle("SayHello FXML");  
    primaryStage.show();  
}
```

5. [Scene Builder](#)

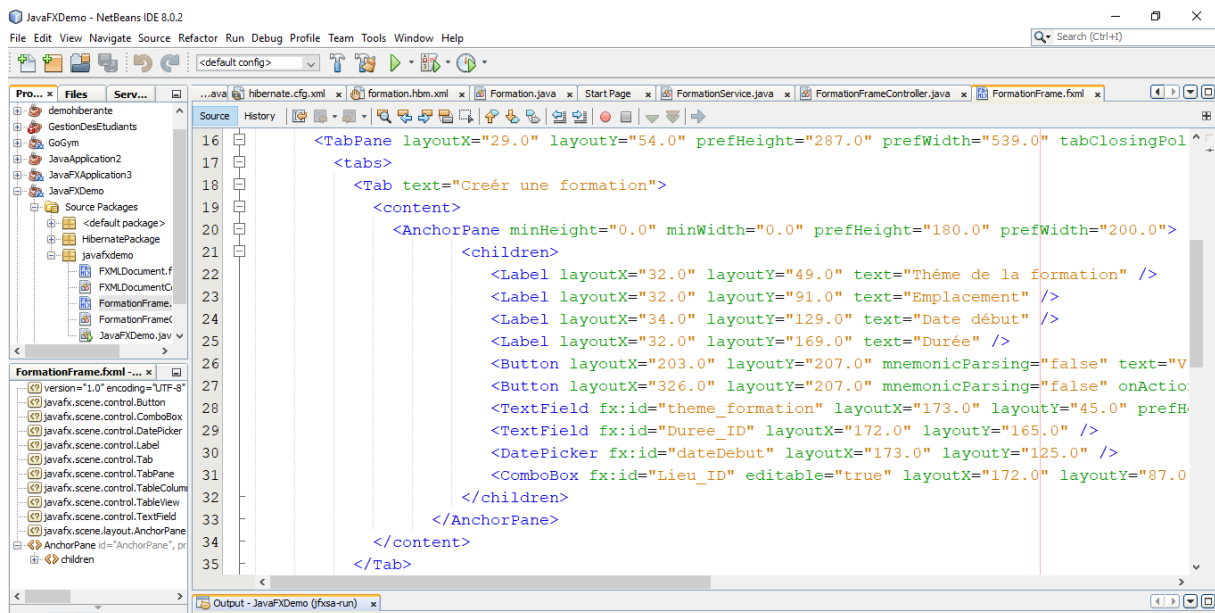
Il est possible de créer les fichiers FXML avec un éditeur de texte mais, plus généralement, on utilise un outil graphique (SceneBuilder) qui permet de concevoir l'interface de manière conviviale et de générer automatiquement le fichier FXML correspondant.

Scene builder étant un plugin pour le design des interfaces graphiques JavaFX en utilisant des drag-and-drop pour la création des composants dans les applications.

Le mode de fonctionnement de cet utilitaire est assez classique avec une zone d'édition centrale, entourée d'un certain nombre d'outils : palettes de conteneurs, de composants, de menus, de graphiques, vues de la structure hiérarchique de l'interface, inspecteurs de propriétés, de layout, etc.



Partie principale du fichier FXML créé avec SceneBuilder :



L'exécution de ce fichier via la classe principale permet de générer l'interface suivante :


Bienvenue à l'application de gestion des formations

EMINES
SCHOOL OF INDUSTRIAL MANAGEMENT
UNIVERSITY MOHAMED VI
POLYTECHNIQUE

Créer une formation Lister les formations

Thème de la formation

Emplacement

Date début 

Durée

Vider Enregistrer

Quitter

III. Première application JavaFX

L'objectif de cette partie est de créer une interface JavaFX pour gérer les clients préalablement étudiées avec Hibernate.

1. Création de l'interface graphique

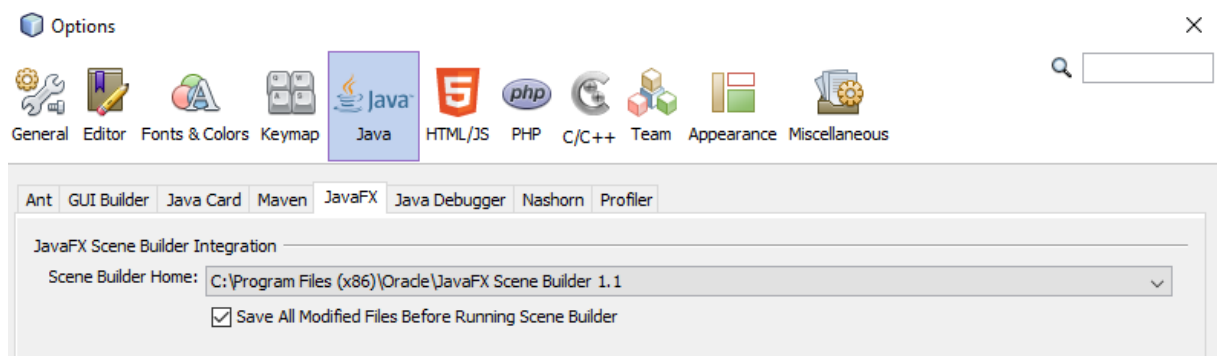
Nous allons aborder dans ce paragraphe la création d'une interface contenant des composants graphiques pour **gérer les clients de la banque** ainsi qu'un composant list.

a) Configuration de Scene Builder

Pour configurer Scene Builder afin de créer des interfaces graphiques en Java FX.

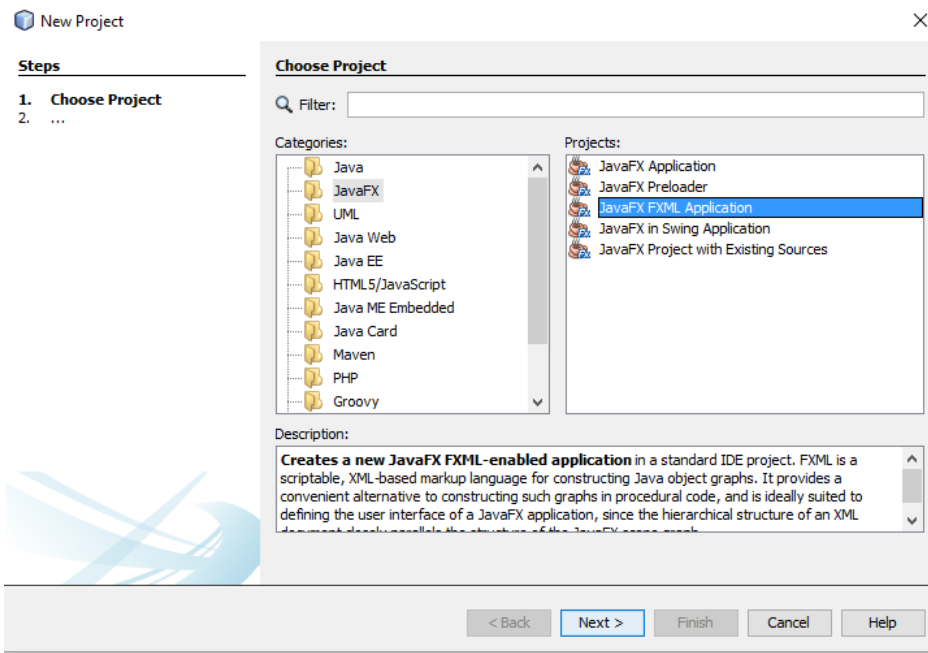
Allez dans NetBeans/tools/options/Java/Javafx et préciser le chemin d'installation de Scene Builder.

NB : si problème de configuration, vous pouvez utiliser directement scene builder.

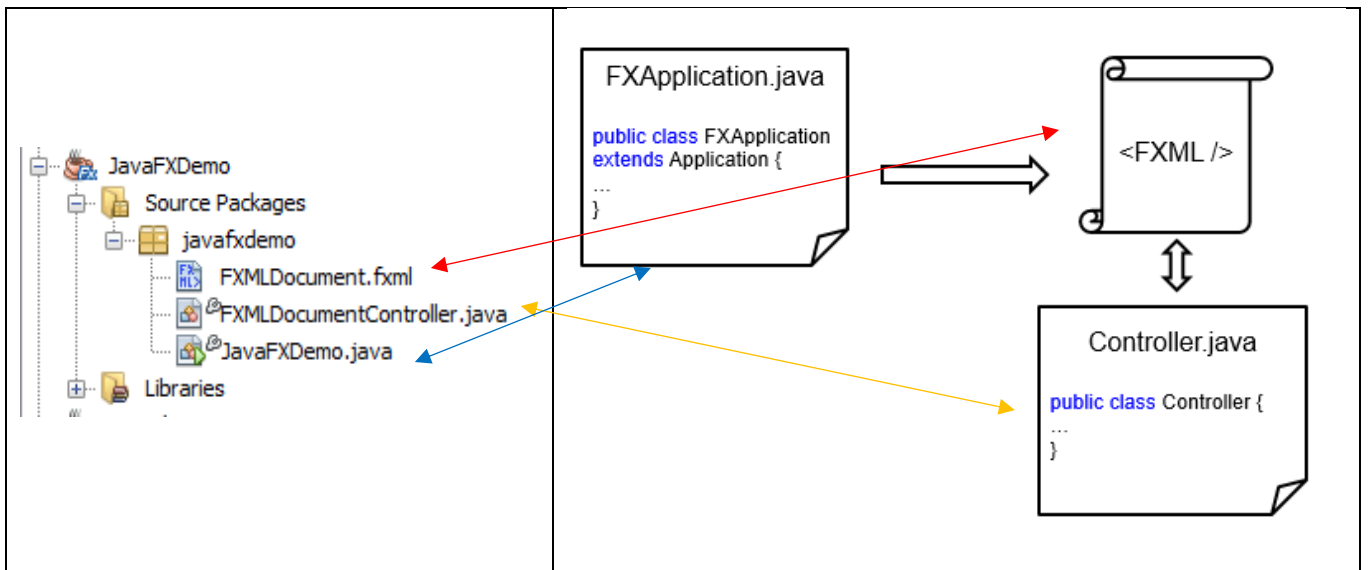


b) Création d'une application JAVAFOX Fxml

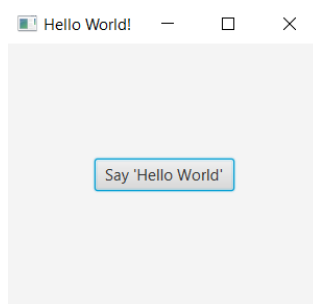
Pour commencer, il faut créer un projet de type « JAVAFOX FXML Application ». Ceci permet de créer un projet en suivant l'architecture définit en haut et contient 3 fichiers (un fichier Fxml, un contrôleur associé à ce fichier et une classe principale pour l'exécution du projet ».



Donner le nom « **GestionBanque** » à votre projet et appuyer sur Finish. Vous allez remarquer la structure de votre application comme suit :



L'exécution du projet à partir de la classe JavaFXApplication.java affiche une interface graphique de ce type.



Le document Fxml contient les éléments suivants :

- un bouton : définit avec la balise « Button » et son id est définit dans l'attribut « fx:id »
- et un libelle : définit par la balise « Label » et son id est définit dans l'attribut « fx:id »

```
FXMLDocument.fxml x
Source History
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import java.lang.*?>
4 <?import java.util.*?>
5 <?import javafx.scene.*?>
6 <?import javafx.scene.control.*?>
7 <?import javafx.scene.layout.*?>
8
9 <AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320" xmlns:fx="http://javafx.com/fxml/1" fx:controller="javafxde
10 <children>
11 <Button layoutX="126" layoutY="90" text="Click Me!" onAction="#handleButtonAction" fx:id="button" />
12 <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69" fx:id="label" />
13 </children>
14 </AnchorPane>
15
```

Quant au contrôleur,

- il a une propriété portant le même nom définit dans l'identifiant de l'élément label et fait référence à cet élément en utilisant l'annotation **@FXML**.
- Et un bouton définit par la balise Button avec l'identifiant button et l'action qui sera exécutée en cliquant sur le bouton est la méthode **handleButtonAction** en utilisant l'annotation **@FXML** définit aussi dans le contrôleur et précédé par le # dans le fichier XML.

```

16  *
17  * @author selwa.elfirdoussi
18  */
19  public class FXMLDocumentController implements Initializable {
20
21      @FXML
22      private Label label;
23
24      @FXML
25      private void handleButtonAction(ActionEvent event) {
26          System.out.println("You clicked me!");
27          label.setText("Hello World!");
28      }
29
30      @Override
31      public void initialize(URL url, ResourceBundle rb) {
32          // TODO
33      }
34
35  }

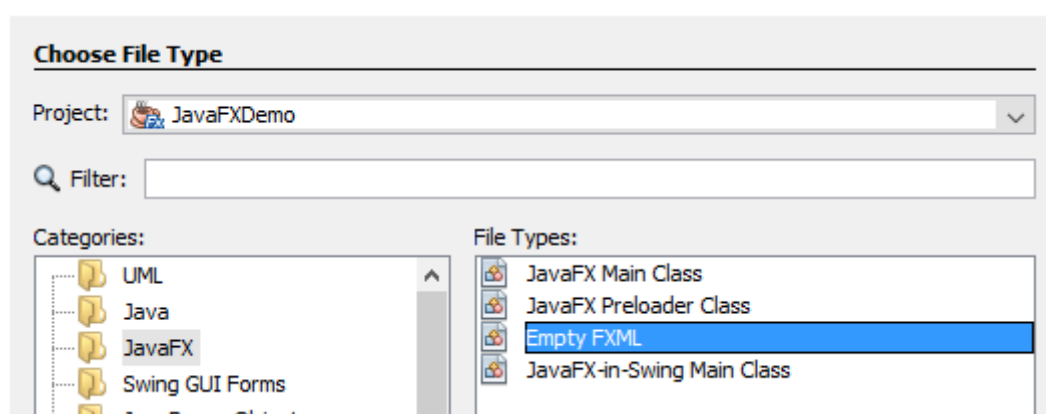
```

c) Création d'une interface de gestion des clients de la banque

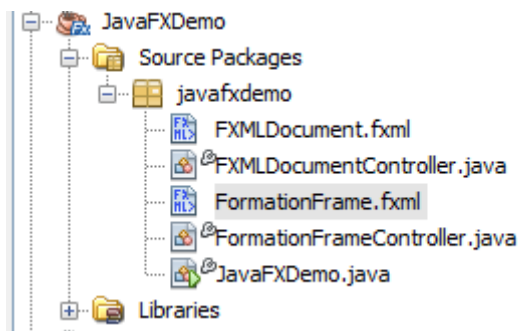
Dans ce qui suit, nous allons modifier l'interface graphique pour :

- Créer des champs de saisie d'un objet **ClientBanque**
- Lister les clients d'une base de données.

Dans le package **GestionBanque**, créer un nouveau fichier de type Fxml et appelé **ClientFrame.fxml** afin de dessiner l'interface.

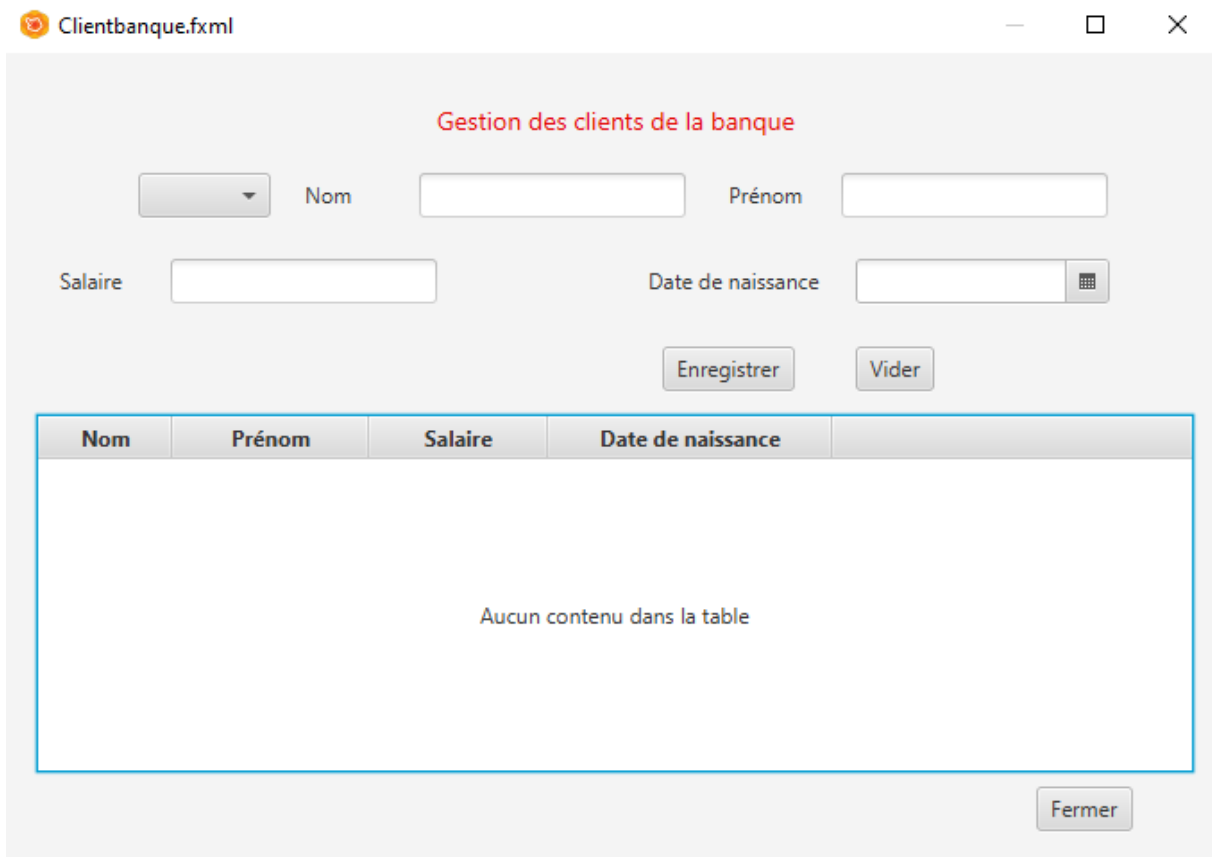


Nommer votre fichier en **ClientFrame.fxml** et cliquer sur Next. NetBeans vous propose de le lier à un controller. Select new create et valider avec le bouton « finish ».



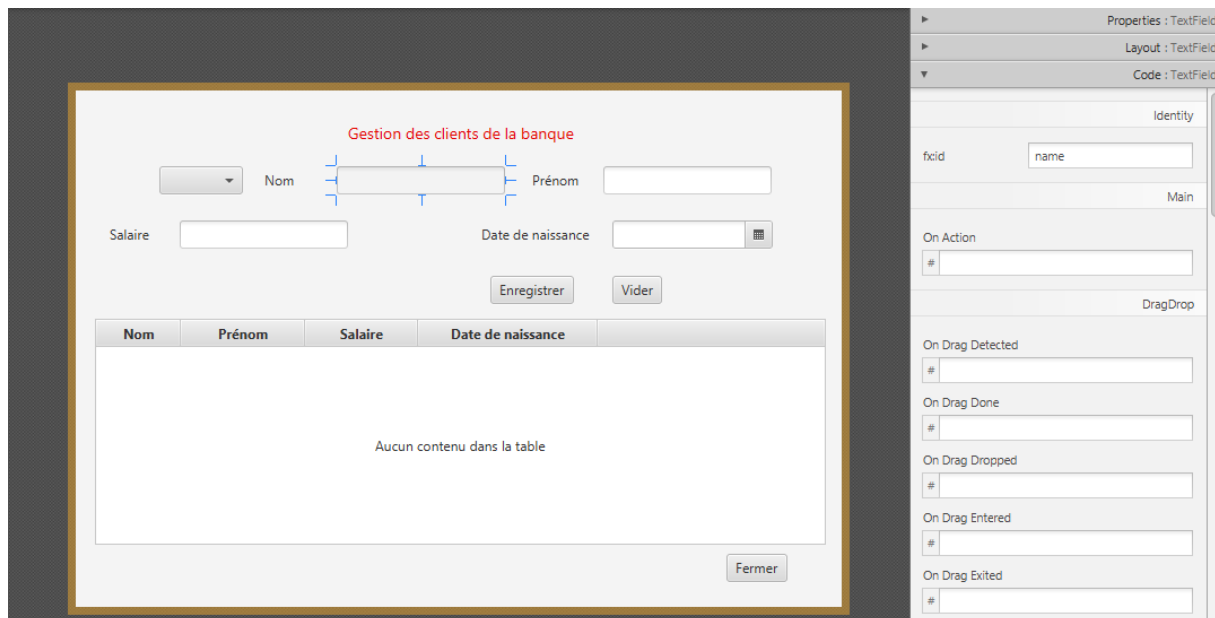
Nous allons poursuivre avec le dessin de l'interface graphique. Cliquer sur le fichier fxml que vous venez de créer. Ceci va l'ouvrir dans l'éditeur Scene Builder. Faire glisser les composants comme suit :

- TabPane : pour les onglets
- Label : pour les libellées
- TextField : pour les zones texte
- DatePicker : pour les dates
- ComboBox : pour les listes déroulantes



Chaque composant contient un ensemble de propriétés accessible à droite de l'interface Scene Builder. Prenons l'exemple du TextField présentant le code. Ces propriétés sont affichées une fois on sélectionne le composant et sont présentés en 3 volets :

- **Propriétés** : TextField : correspond aux informations d’affichage
- **Layout** : TextField : correspond à la taille et au positionnement du composant
- **Code** : TextField : correspond à la partie codage du composant comme par exemple la définition de l’attribut **fx:id** et les actions.



Je vous invite à sauvegarder et à éditer votre document fxml via un click-droit edit, vous allez remarquer que Scene Builder a intégré toutes les balises correspondantes aux différents composants que vous avez dessinés.

```
<AnchorPane id="AnchorPane" prefHeight="457.0" prefWidth="678.0" xmlns:fx="http://javafx.com/fxml/1"
  <children>
    <TableView fx:id="ListClient" layoutX="17.0" layoutY="202.0" prefHeight="200.0" prefWidth="64
      <columns>
        <TableColumn fx:id="cltName" prefWidth="75.0" text="Nom" />
        <TableColumn fx:id="cltFirstName" prefWidth="110.0" text="Prénom" />
        <TableColumn fx:id="cltSalaire" prefWidth="100.0" text="Salaire" />
        <TableColumn fx:id="cltBirthDate" prefWidth="159.0" text="Date de naissance" />
      </columns>
    </TableView>
    <TextField fx:id="name" layoutX="231.0" layoutY="67.0" />
    <TextField fx:id="firstName" layoutX="467.0" layoutY="67.0" />
    <TextField fx:id="salaire" layoutX="92.0" layoutY="115.0" />
    <DatePicker fx:id="birthDate" layoutX="475.0" layoutY="115.0" prefHeight="25.0" prefWidth="14
    <Label layoutX="167.0" layoutY="71.0" text="Nom" />
    <Label layoutX="404.0" layoutY="71.0" text="Prénom" />
    <Label layoutX="30.0" layoutY="119.0" text="Salaire" />
    <Label layoutX="359.0" layoutY="119.0" text="Date de naissance" />
    <Button fx:id="saveButton" layoutX="367.0" layoutY="164.0" mnemonicParsing="false" text="Enre
```

2. Affichage de l'interface graphique via la classe principale

A ce stade, nous avons créé l'interface graphique avec des champs de saisie et des boutons. Nous allons charger cette interface dans la classe principale pour l'exécuter. Pour cela, nous allons la charger dans la classe « Gestionbanque.java » en modifiant la méthode start(). Rappelez-vous que la méthode start permet de charger une interface via son fichier Fxml et la démarrer. Modifiez cette méthode en précisant le chemin du fichier fxml à charger et qui concerne « clientbanque.fxml ».

```
public class GestionBanque extends Application {  
    @Override  
    public void start(Stage stage) throws Exception {  
        Parent root = FXMLLoader.load(getClass().getResource("Clientbanque.fxml"));  
  
        Scene scene = new Scene(root);  
  
        stage.setScene(scene);  
        stage.show();  
    }  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

Exécuter le projet : ceci affiche l'interface que vous venez de créer.

NB : si vous cliquer sur les boutons, rien ne se passe.

3. Gestion des évènements : lier un composant à une action

2.1 Renseigner les éléments dans la combox

Le champ civilité est un champ de type liste déroulante présenté par le composant ComboBox. Afin de le remplir, on peut le faire via la méthode Initialize de la classe controller.

Chaque contrôleur contient une méthode Initialize() permettant d'exécuter un ensemble d'instructions à l'ouverture d'une interface.

```
@Override
public void initialize(URL url, ResourceBundle rb) {
    //Ajouter les éléments dans la comboBox Civilite
    civilite.getItems().addAll("Mr", "Mme");
    civilite.getSelectionModel().select("Mr");
}
```

2.2 Créer des actions pour les boutons enregistrer et fermer

L'objectif de cette partie est de lier le bouton « enregistrer » et « Fermer » à une action. En effet, après que l'utilisateur renseigne les informations sur le client, le bouton enregistrer doit permettre d'insérer ce client dans la base de données. Rappelez-vous que nous avons fait la même chose dans le cours d'hibernate. Donc nous allons réutiliser ce code pour définir l'action de sauvegarde.

- Configurer Hibernate dans votre projet JavaFX (fichier de configuration, la classe de persistance, et le fichier de mapping) : vous pouvez les copier directement de votre ancien projet.
- Créer la couche Service qui contiennent les méthodes de création d'un nouveau client.

```
public static int createClient(Client clt) {
    SessionFactory sessionFact = HibernateUtil.getSessionFactory();
    session = sessionFact.openSession();
    Transaction tx = session.beginTransaction();
    int idClient = (int) session.save(clt);
    tx.commit();
    session.close();
    return idClient;
}
```

2.3 Action d'enregistrement :

L'enregistrement se fait via le bouton « Enregistrer ». Pour lier ce bouton à une action, il faut le configurer dans le fichier « .fxml »

Dans la balise <Button> permettant de créer le bouton « Enregistrer », rajouter un attribut OnAction= « #saveClient ». Pour commencer la méthode sera surlignée en rouge étant donné que nous ne l'avons pas encore définie dans le Controller.

Aller dans le controller « ClientBanqueController.java » et ajouter une méthode avec la signature suivante :

```
<Button fx:id="saveButton" layoutX="367.0" layoutY="164.0" mnemonicParsing="false"
text="Enregistrer" onAction="#saveClient"/>
```

C'est dans cette méthode que nous allons créer un client en utilisant les données saisies par l'utilisateur.

```
@FXML
public void saveClient(ActionEvent event) {

}
```

Pour ce faire, il faut commencer par lier les champs text dans le contrôleur afin de récupérer leurs valeurs. Ceci se fait via l'annotation @FXML et en les définissant en tant qu'attribut de la classe « ClientBanqueController »

```
@FXML
private ComboBox civilite;
@FXML
private TextField name;
@FXML
private TextField firstName;
@FXML
private TextField salaire;
@FXML
private DatePicker birthDate;
```

La récupération des valeurs saisies par l'utilisateur se fait via la méthode `getText()` de l'objet `textField`.

Pour les listes déroulantes, la récupération des valeurs se fait par la méthode
« `civilite.getSelectionModel().getSelectedItem()` »

Pour les dates, on peut utiliser la méthode `ValueOf` de la classe `java.sql.date` pour avoir un objet `Date` à partir du `DatePicker`

```
java.sql.Date.valueOf(DateDebut.getValue())
```

Enfin, le salaire est un champ texte dans l'interface et l'attribut de la classe `Client` est un type `Float`. Donc il faut convertir le type `String` à `float`. Pour se faire la classe `Float` contient une méthode `parseFloat` qui prend comme paramètre un objet `String` et le convertit en `float` comme suit :

```
Float.parseFloat(salaire.getText())
```

De même pour le cas de `int` « `Integer.parseInt(String)` »

NB : toutes les classes importées doivent être celles de l'API JAVA FX et non pas swing ou awt.

L'action d'enregistrement sera comme suit :

```
@FXML
public void saveClient(ActionEvent event) {
    Client clt = new Client();
    clt.setNomClient(name.getText());
    clt.setPrenomClient(firstName.getText());
    clt.setSalaire(Float.parseFloat(salaire.getText()));
    ClientService.createClient(clt);
}
```

3.1 Action de fermeture :

L'objectif étant de fermer la fenêtre en cliquant sur le bouton « Fermer » avec un message d'alerte. De la même manière, lier le bouton à une action « closeWindow » dans le fichier fxml.

Créer cette méthode au niveau du contrôleur avec l'annotation @FXML

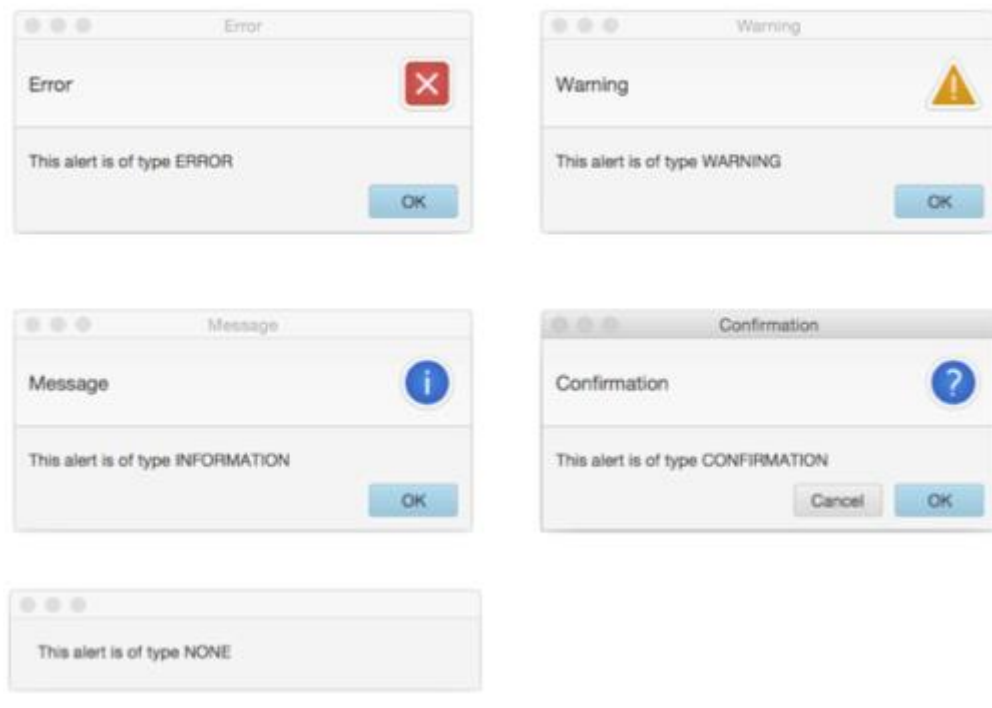
Déclarer le bouton dans les attributs du contrôleur avec l'annotation @FXML

En JavaFx, la gestion des messages alertes se fait via la classe Alert. Lors de la création d'un objet de type Alert, on définit son type :

```
@FXML
private Button CloseButton;

@FXML
private void closeFrame(){
    Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
    alert.setTitle("Confirmation dialog");
    alert.setHeaderText("Voulez vous fermer la fenetre? ");
    Optional<ButtonType> result = alert.showAndWait();
    if (result.get() == ButtonType.OK){
        Stage stage = (Stage) CloseButton.getScene().getWindow();
        stage.close();
    }
}
```

- `AlertType.INFORMATION`
- `AlertType.WARNING`
- `AlertType.ERROR`
- `AlertType.CONFIRMATION`



NB : la méthode `getScene` d'un composant javafx permet de récupérer la scene du composant et ainsi que la fenêtre via `getWindow` afin de lancer des actions sur cet objet (exemple : `close()`).

```
Stage stage = (Stage) CloseButton.getScene().getWindow();
```

```
stage.close();
```

```
@FXML
```

```
public void closeClient(ActionEvent event) {
```

```
    Stage stage = (Stage) closeButton.getScene().getWindow();
```

```
    stage.close();
```

```
}
```

4. Gestion des listes en JAVAFX : TableView

Nous allons définir dans l'interface graphique une table permettant de lister tous les clients existants dans la table « ClientBanque » de la base de données.

- a) Créer une méthode dans la classe ClientService de type static qui rend une liste des clients en utilisant la méthode createQuery.

NB : le mot clé static permet d'appeler une méthode directement à partir de la classe sans avoir à instancier un objet.

```
public static List listClient() {
    SessionFactory sessionFact = HibernateUtil.getSessionFactory();
    Session session = sessionFact.openSession();
    List listClient = session.createQuery("from Client ").list();
    return listClient;
}
```

4.1 Utilisation des Tables pour afficher les clients par colonne :

JavaFX fournit une classe **TableView**, il est utilisé avec TableColumn pour vous aider à afficher les données sous forme de tableau (forme Tabulaire).

En utilisant SceneBuilder, insérer un nouvel onglet « listClients » et dessiner un composant « Table View » contenant les colonnes pour afficher systématiquement l'identifiant, Civilité, Nom, Prénom, date de naissance et Salaire. Pour chaque colonne, préciser sont `fx:id` comme suit :

```
<TableView fx:id="ListClient" layoutX="17.0" layoutY="202.0" prefHeight="200.0"
  <columns>
    <TableColumn fx:id="cltId" prefWidth="80" text="Identifiant" />
    <TableColumn fx:id="cltName" prefWidth="100" text="Nom" />
    <TableColumn fx:id="cltFirstName" prefWidth="110.0" text="Prénom" />
    <TableColumn fx:id="cltSalaire" prefWidth="100.0" text="Salaire" />
  </columns>
</TableView>
```

Rappelons que dans chaque contrôleur contient une méthode `Initialize()` permettant d'exécuter un ensemble d'instructions à l'ouverture d'une interface.

Nous allons maintenant charger la liste des clients dans cette table, dans la méthode, `initialize` du contrôleur afin d'avoir la liste au moment de l'affichage de l'interface.

Déclarer la tableview dans le contrôleur en utilisant l'annotation `@FXML` ainsi que ses colonnes.

La déclaration des colonnes se fait en précisant le nom de la classe et le type de l'attribut de la classe.


```
@FXML
private TableView ListClient;
@FXML
private TableColumn cltId;
@FXML
private TableColumn cltName;
@FXML
private TableColumn cltFirstName;
@FXML
private TableColumn cltSalaire;
```

TableView n'a absolument aucune idée de comment trouver les valeurs à afficher dans ses lignes pour chaque colonne. Pour chacune des colonnes, il va donc falloir spécifier une CellValueFactory qui, à partir d'un objet de type Client présent sur une ligne donnée, va nous retourner une valeur du type approprié pour cette colonne comme suit :

```
@Override
public void initialize(URL url, ResourceBundle rb) {
    //Ajouter les éléments dans la comboBox Civilite
    civilite.getItems().addAll("Mr", "Mme");
    civilite.getSelectionModel().select("Mr");

    //charger la liste des clients dans la table view listClient
    java.util.List listClients = ClientService.listClient();
    cltId.setCellValueFactory(new PropertyValueFactory("idClient"));
    cltName.setCellValueFactory(new PropertyValueFactory("nomClient"));
    cltFirstName.setCellValueFactory(new PropertyValueFactory("prenomClient"));
    cltSalaire.setCellValueFactory(new PropertyValueFactory("salaire"));
    ListClient.getItems().addAll(listClients);
}
```

A l'exécution du projet, le contrôleur charge la liste des clients en précisant pour chaque colonne la valeur correspondante.

Gestion des clients de la banque

Mr Nom Prénom

Salaire Date de naissance

Identifiant	Nom	Prénom	Salaire
2	HRIMECH	HAMID	10000.0
3	ESSABATR	DRISS	13000.0
4	AMAL	TEST	12000.0
5		TEST 2	14000.0
6		test 5	16000.0
7	Berrada	youssef	20000.0
8			

4.2 Modification et suppression d'un client en JAVAFX à partir de la table

Rappelez-vous que vous avez créé dans la classe ClientService une méthode qui permet de modifier un objet client et un autre qui le supprime.

```
public static void updateClient(Client clt) {
    Transaction tx = session.beginTransaction();
    session.update(clt);
    tx.commit();
    session.close();
}

public static void deleteClient(Client clt) {
    SessionFactory sessionFact = HibernateUtil.getSessionFactory();
    Session session = sessionFact.openSession();
    Transaction tx = session.beginTransaction();
    session.delete(clt);
    tx.commit();
    session.close();
}
```

Nous allons modifier l'interface pour ajouter un bouton de suppression et de modification d'un client.

Identifiant	Nom	Prénom	Salaire	
2	HRIMECH	HAMID	10000.0	
3	ESSABATR	DRISS	200000.0	
8	Berrada	youssef	20000.0	
9	Malik	Youssef	45000.0	
10	Hamid	Ilham	12000.0	
11	karim	hassan	18000.0	
12	karim	Hellam	50000.0	

A noter que la récupération de l'objet sélectionné dans une tableView se fait à l'aide de la méthode :

```
Client cltToDel = (Client) ListClient.getSelectionModel().getSelectedItem();
```

Créer les deux méthodes relatives aux actions des boutons ajoutés dans la classe contrôleur.

a) Cas de la suppression :

L'action relative à la suppression permettra de récupérer l'objet sélectionné et appel de la méthode delete avec une alerte de confirmation si on veut.

b) Cas de la modification :

Il existe deux manières pour modifier les attributs d'un client:

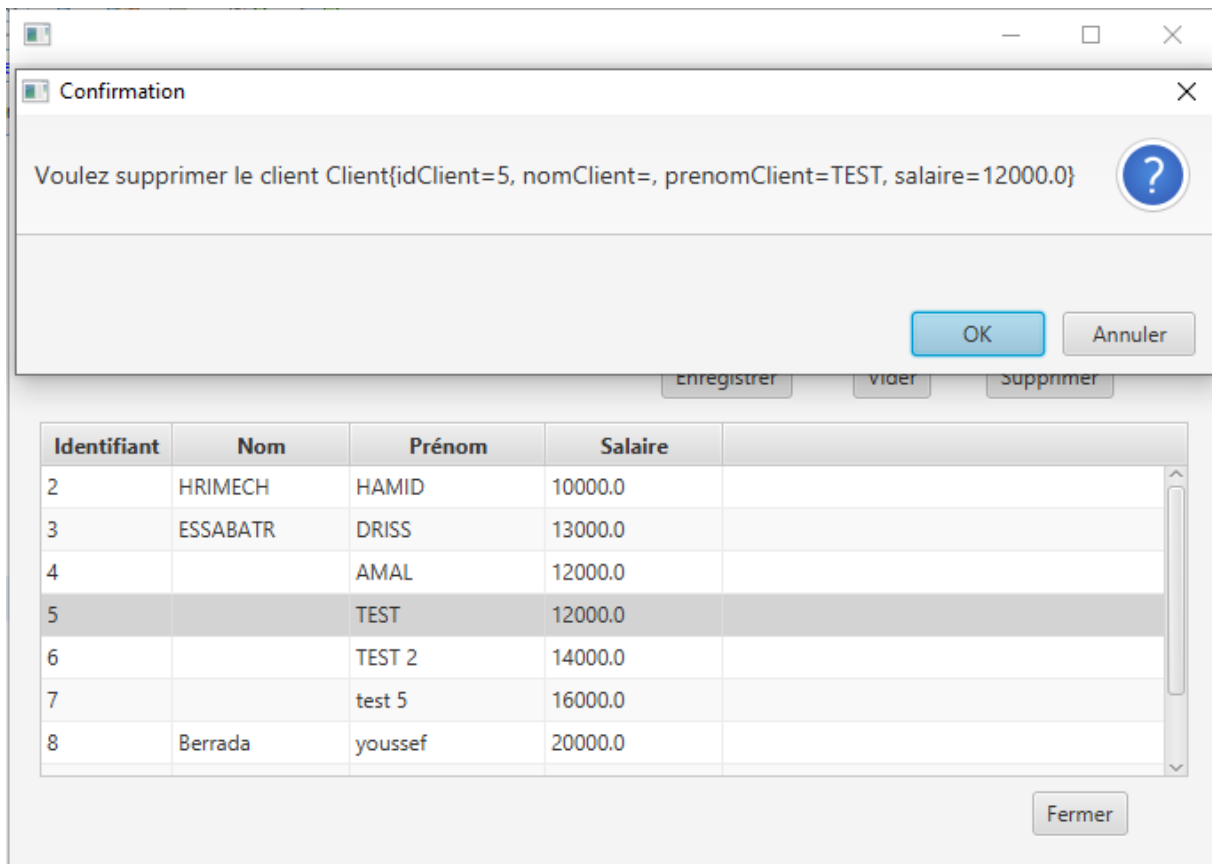
- Le faire directement dans la liste
- Les renseigner dans les champs correspondants et cliquer sur Enregistrer (attention cette fois, en précisant l'identifiant)

Suppression d'un client

Dans le fichier FXML

```
<Button layoutX="550.0" layoutY="164.0" mnemonicParsing="false" text="Supprimer"
onAction="#deleteClient"/>
```

```
@FXML
public void deleteClient(ActionEvent event){
    Client cltToDel = (Client) ListClient.getSelectionModel().getSelectedItem();
    Alert alertDel = new Alert(Alert.AlertType.CONFIRMATION);
    alertDel.setHeaderText("Voulez supprimer le client "+ cltToDel);
    Optional<ButtonType> result = alertDel.showAndWait();
    if (result.get() == ButtonType.OK){
        ClientService.deleteClient(cltToDel);
        ListClient.getItems().remove(cltToDel);
    } else {
        closeClient(event);
    }
}
```



Modification dans la liste :

Pour ce faire il faut que les cellules de la table view soit éditables. Pour cela, allez dans la méthode initialize et ajouter la possibilité d'éditer les cellules via la méthode setEditable(true) de la classe TableView.

Chaque colonne de la table dispose de sa propre propriété cellFactory qui permet de placer un callback permettant de générer des cellules pour chaque ligne de cette colonne.

```
ListClient.setEditable(true);
cltFirstName.setCellFactory(TextFieldTableCell.forTableColumn());
cltSalaire.setCellFactory(TextFieldTableCell.forTableColumn(new FloatStringConverter()));
civillite.setCellFactory(ComboBoxTableCell.forTableColumn("Mr", "Mme"));
```

Créer dans le contrôleur deux méthodes qui permettent de :

- Récupérer un objet de type Client et qui est sélectionné dans la liste
- Modifier les champs sélectionnés en récupérant la nouvelle valeur de la cellule concernée
- Appeler la méthode Update de l'objet pour mettre à jour dans la base de données

```
@FXML
public void editFirstNameFromList(TableColumn.CellEditEvent<Client, String>
    ClientFirstNameEvent) {
    Client cltToEdit = (Client) ListClient.getSelectionModel().getSelectedItem();
    cltToEdit.setPrenomClient(ClientFirstNameEvent.getNewValue());
    ClientService.updateClient(cltToEdit);
}
```

```
@FXML
public void editSalaireFromList(TableColumn.CellEditEvent<Client, Float>
    ClientSalaireEvent) {
    Client cltToEdit = (Client) ListClient.getSelectionModel().getSelectedItem();
    cltToEdit.setSalaire(ClientSalaireEvent.getNewValue());
    ClientService.updateClient(cltToEdit);
}
```

Lier ces méthodes aux colonnes concernées dans le fichier fxml directement ou via Scene Builder dans l'action onEditCommit pour chaque TableColumn qui correspond.

```
<columns>
  <TableColumn fx:id="cltId" prefWidth="80" text="Identifiant" />
  <TableColumn fx:id="cltName" prefWidth="100" text="Nom" />
  <TableColumn fx:id="cltFirstName" text="Prénom" onEditCommit="#editFirstNameFromList" />
  <TableColumn fx:id="cltSalaire" text="Salaire" onEditCommit="#editSalaireFromList"/>
</columns>
```

Les propriétés importantes propres à ce contrôle sont les suivantes :

- cellFactory - de type Callback<ListView<V>, ListCell<V>> ; une fabrique à cellules qui est utilisée pour produire une cellule destinée à afficher chaque valeur visible de la liste ;
- editable - permet de spécifier si les lignes de la liste peuvent être éditées ;
- editingIndex - une propriété en lecture seule qui retourne l'indice de la ligne en train d'être éditée ;
- items - une ObservableList<V> contenant les objets à afficher ;
- onEditCancel - un callback appelé lorsque l'utilisateur annule son édition ;
- onEditCommit - un callback appelé lorsque l'utilisateur valide son édition ;
- onEditStart - un callback appelé lorsque l'utilisateur démarre son édition ;

Modification dans les champs de création

Rappelons que tout au début, l'interface graphique de saisir les attributs d'un objet Client. Nous allons utiliser ce même onglet pour remplir les valeurs pour chaque attribut par celui qui a été sélectionné dans la table. Ceci se fait via la méthode `setText()` de l'objet `TextField`.

A ce stade, il faut modifier le code de la méthode « `saveClient` » pour modifier un client dans le cas si un identifiant existe.

- Ajouter un bouton « Modifier » qui permet de renseigner les champs sélectionnés dans la table dans les zones textes correspondant.
- Modifier la méthode `SaveClient` pour remplir ce champ avec la méthode `getId()` de la sélectionnée et faire un update du client si `id <> null` et create si `id = null`