

# NDC : Développement d'application

## Le Framework JavaFX



**Email:** [kahina.hassam@yncrea.fr](mailto:kahina.hassam@yncrea.fr); [vincent.lefevere@yncrea.fr](mailto:vincent.lefevere@yncrea.fr);  
[carol.habib@yncrea.fr](mailto:carol.habib@yncrea.fr)

**Bureau:** T336, T342

**Département:** Organisation , Management et Informatique

# Plan du cours

- **Installation**
- Programmation Java
- Le FXML
- Utilisation d'un RAD
- Disposition adaptative
- Traitement des événements
- Application multifenêtres
- Annexes

# Installation d' Eclipse Normalement déjà fait

- Téléchargez et installez le jdk version 11.
- Téléchargez Eclipse version (la toute dernière version)
- Dé-zippez Eclipse vers un répertoire d'installation.

# Installation de SceneBuilder

Téléchargez « SceneBuilder » depuis l'URL :

<http://gluonhq.com/products/scene-builder/>

Installez-le.

Repérez le chemin de l'exécutable

normalement sous Windows cela peut être :

C:\Program Files\SceneBuilder\SceneBuilder.exe

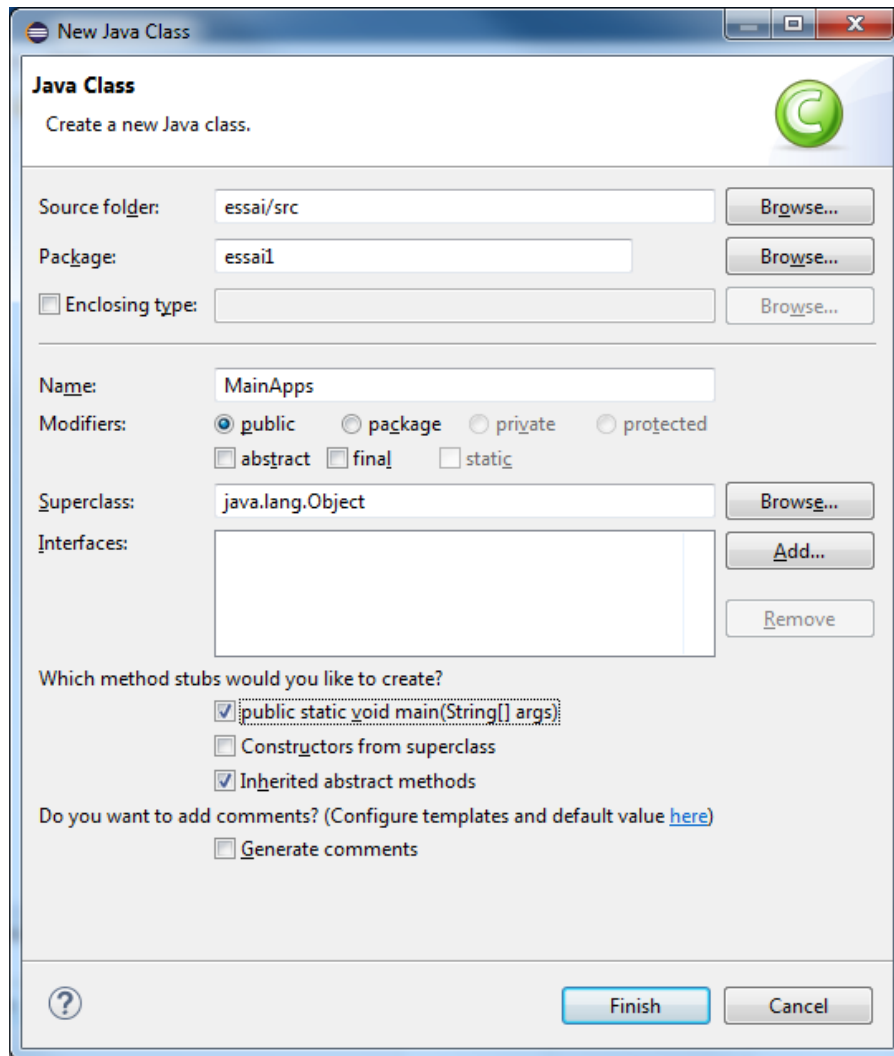
# Installation de JavaFx dans Eclipse:

- Pour les étapes d'installation du Javafx dans Eclipse voir la vidéo d'installation

# Plan du cours

- Introduction
- **Programmation Java**
- Le FXML
- Utilisation d'un RAD
- Disposition adaptative
- Traitement des événements
- Application multifenêtres
- Annexes

# Création d'un projet JavaFX



## Méthode I

Cela se crée comme un projet Java.

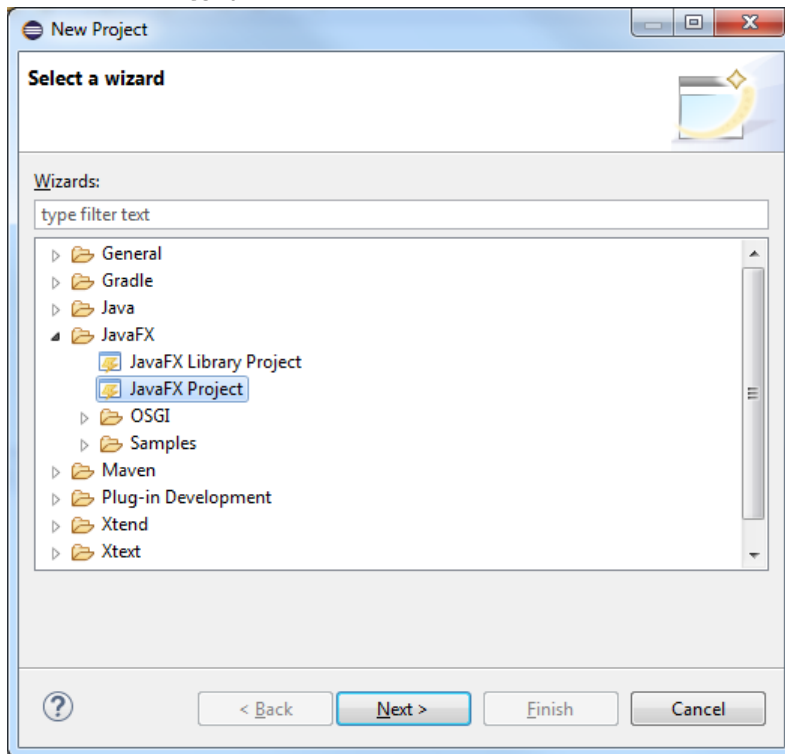
Ensuite on crée une classe que l'on nommera « MainApp » comme point de lancement de l'application graphique (On prend ce nom par convention mais un autre nom pourrait être utilisé).

Remarque : On peut avoir plusieurs « MainApp » dans un même projet si on les stocke dans des « Package » différents.

# La création d'un projet JavaFX

## Méthode 2

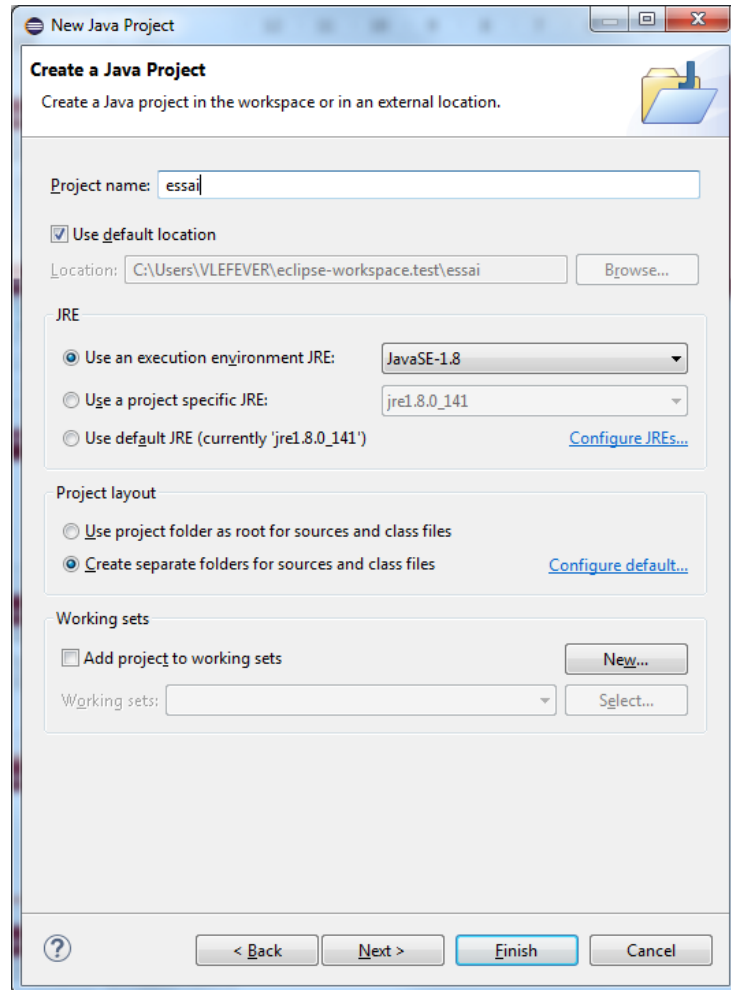
Eclipse peut également créer directement un projet JavaFX en créant les fichiers de base et en plaçant dans ces fichiers le code minimal.



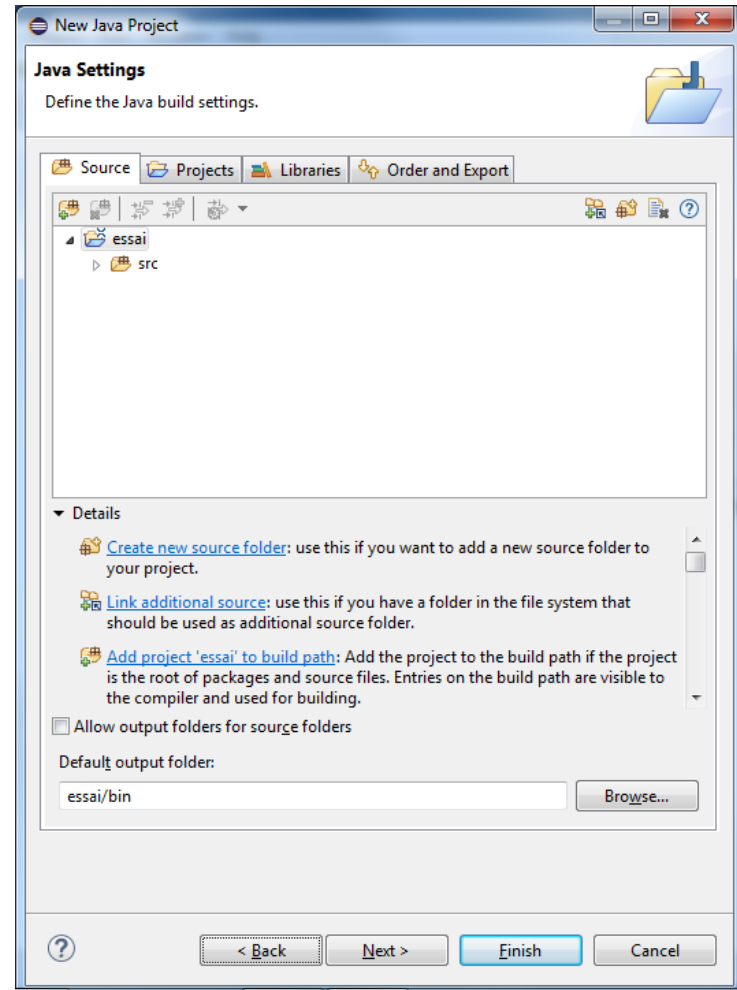
On choisit dans le menu « File / New » l'item « Project » au lieu de « Java Project ». Et la fenêtre ci contre s'ouvre pour nous laisser définir le type de projet souhaité. On sélectionne dans « JavaFX » « JavaFX Project ».



# La création d'un projet JavaFX



On clique sur Next



On clique sur Next



# La création d'un projet JavaFX

New Java Project

Application type: Desktop

Package Name: application

Declarative UI

Language: None

Root-Type: javafx.scene.layout.BorderPane

File Name: Sample

Controller Name: SampleController

< Back Next > Finish Cancel

On change les valeurs



New Java Project

Application type: Desktop

Package Name: application

Declarative UI

Language: FXML

Root-Type: javafx.scene.layout.AnchorPane

File Name: Sample

Controller Name: SampleController

? < Back Next > Finish Cancel

On clique sur Finish



# La création d'un projet JavaFX

On récupère ainsi dans le package « application » :

- un fichier « Main.java » équivalent de notre fichier « MainApp.java » (contient le code de chargement),
- un fichier « SampleController.java » équivalent de notre fichier « Controller.java » (classe vide),
- un fichier « Sample.fxml » équivalent de notre fichier « RootLayout.fxml » (contient le conteneur choisi comme racine),
- et un fichier « application.css » comme fichier feuille de style (vide).

À voir dans la suite du chapitre

# Création d'un projet JavaFX

On ajoute à la ligne « `public class MainApp {` », « **`extends Application`** » qui devient ainsi « `public class MainApp extends Application {` »

Le mot clef « **`extends`** » signifie que la classe « `MainApp` » va hériter :

- ✓ Du code des méthodes de la classe « `Application` » de JavaFx.
- ✓ D'obligation d'écrire certaines méthodes appelées.

Il faut également indiquer depuis quel package, on importe la classe « `Application` » en ajoutant dans la zone des « `import` » la ligne suivante :

**`import javafx.application.Application;`**

Une erreur nous indique qu'il faut définir une méthode :

**`public void start(Stage primaryStage) throws Exception`**

« `Stage` » est le type d'objet JavaFX passé à la méthode `start`. On doit donc ajouter :

**`import javafx.stage.Stage;`**

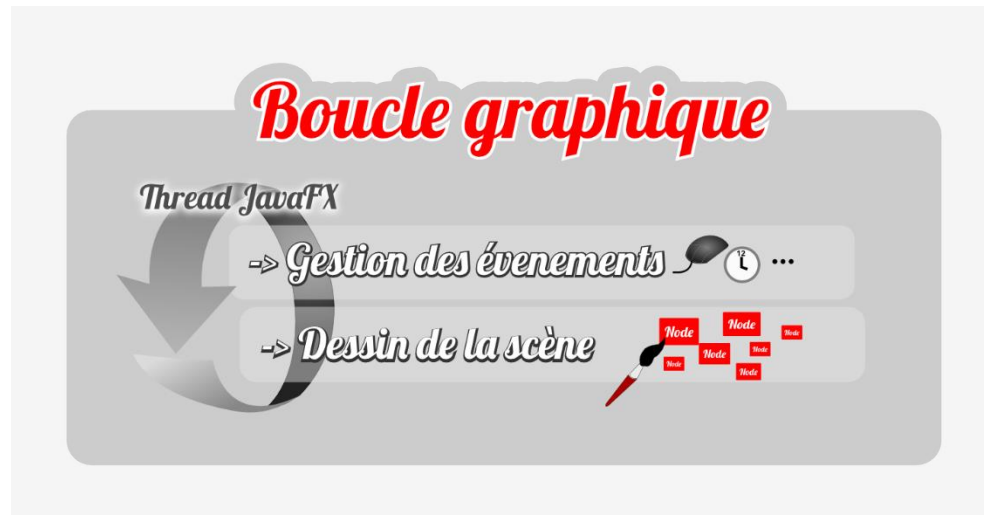
# Création d'un projet JavaFX

On a obtenu un programme JavaFX qui ne fait rien car le main ne contient pas d'instruction de lancement d'une interface graphique JavaFX.

Il faut ajouter dans le main l'appel de la méthode héritée de la classe « Application » :

*launch(args);*

Remarque : cette méthode initialise l'environnement graphique JavaFX et appelle la méthode « start » et lance une boucle de gestion des événements (qui proviennent de l'IHM)



# Structure de la fenêtre JavaFX

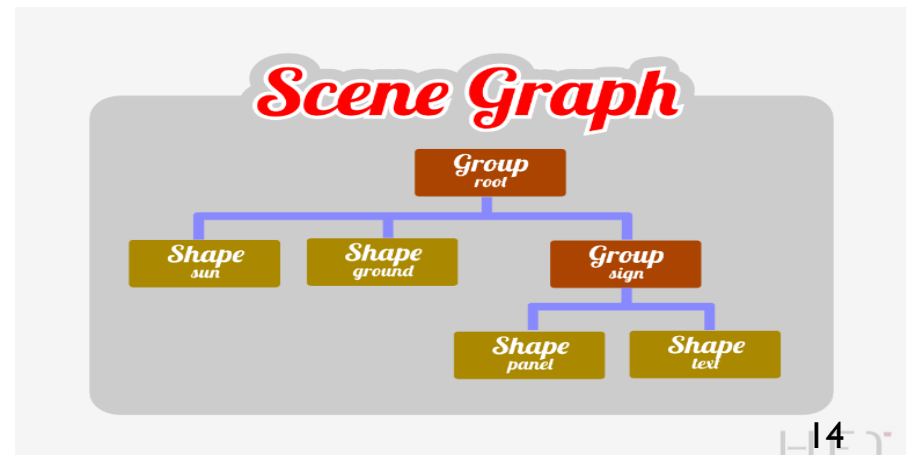
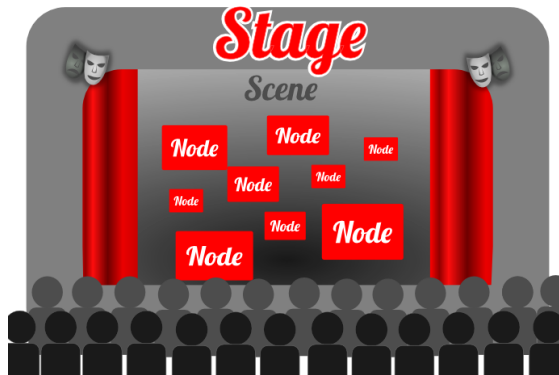
Les termes de JavaFX sont construits par analogie avec un théâtre.

Il y a, en premier lieu, une estrade ou « Stage » en anglais (et donc en JavaFX). C'est le lien avec le système de fenêtre des systèmes d'exploitation.

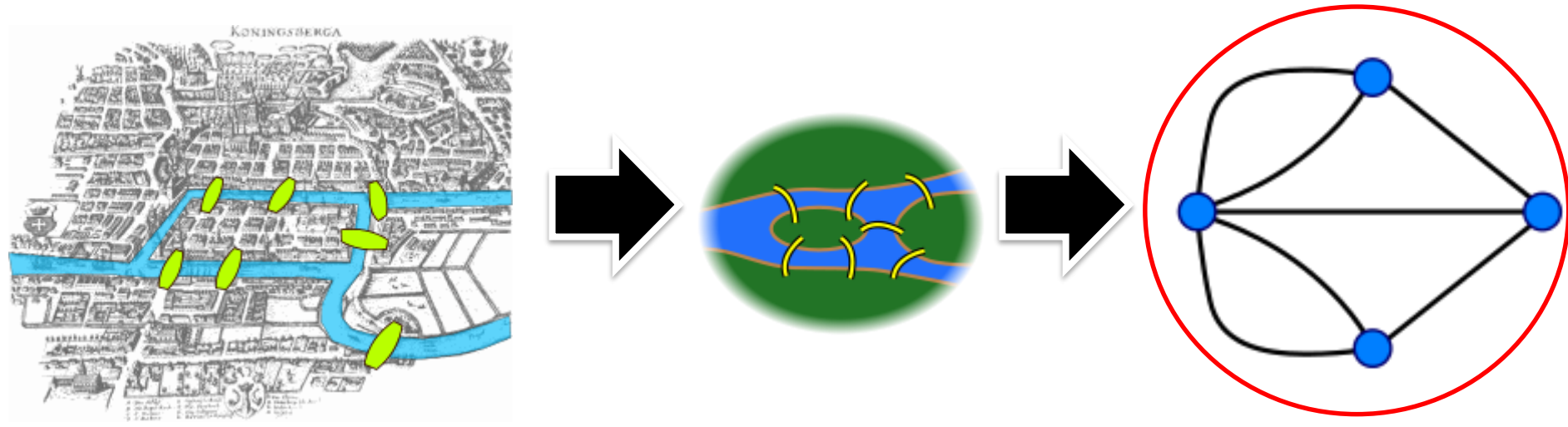
Il y a, en second lieu, le décor ou « Scene » en anglais (et donc en JavaFX).

On doit donc assigner à l'estrade un décor.

En troisième lieu, on place des objets graphiques regroupés dans une **structure arborescente** que l'on utilise pour fabriquer le décor.



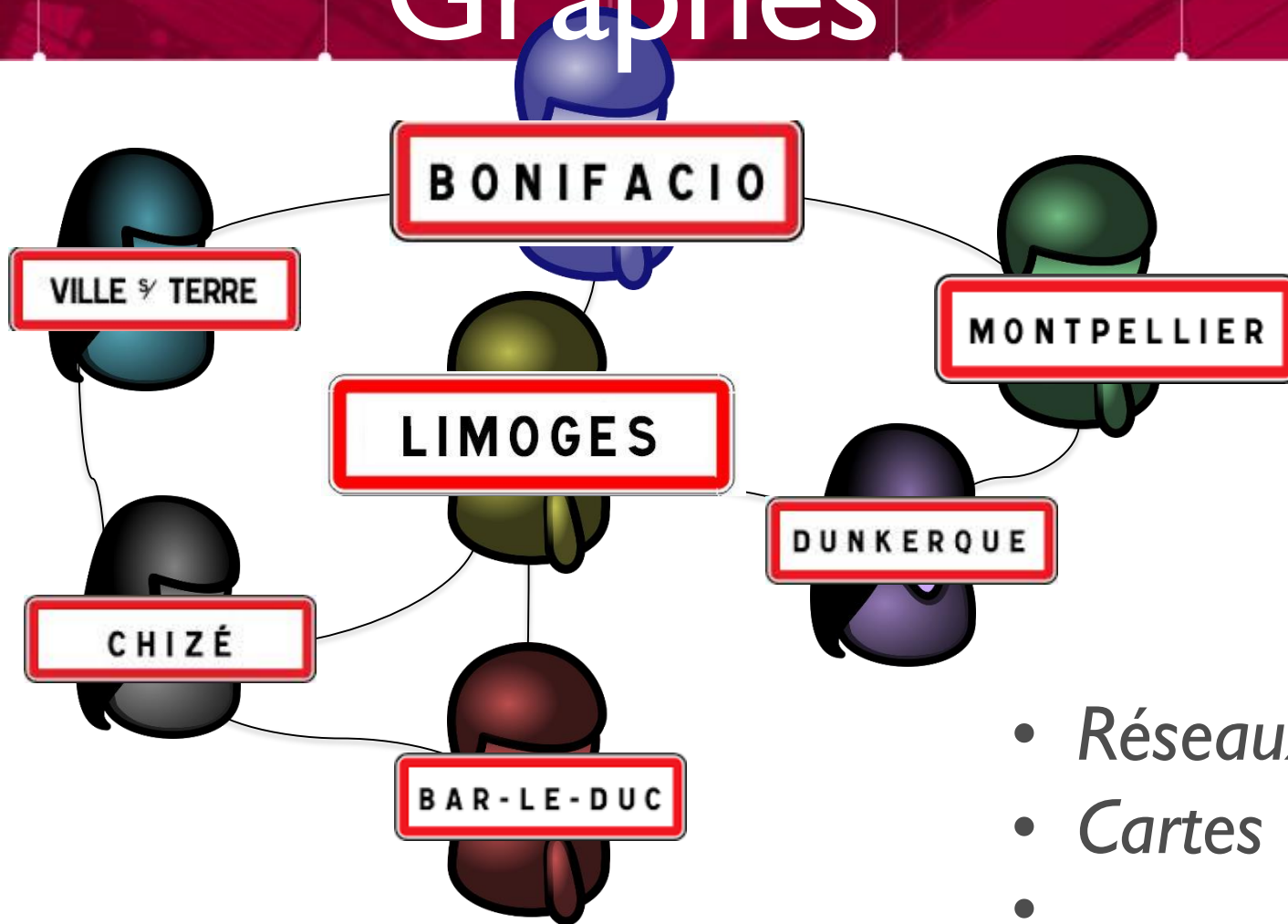
# Structure arborescente Graphes



- Représentation standardisée sous forme de ronds reliés par des traits.
- Les ronds représentent n'importe quoi, les traits représentent les transitions possibles



# Structure arborescente Graphes

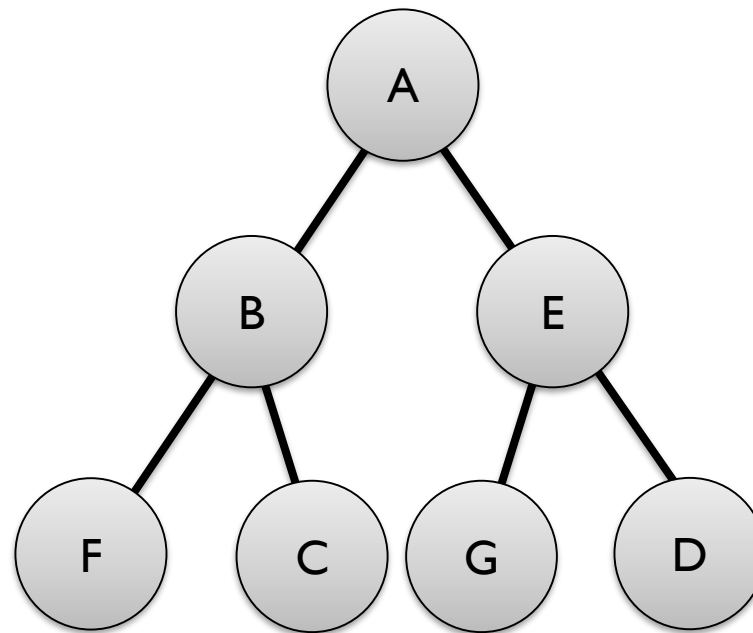


- Réseaux sociaux
- Cartes
- ...



# Structure arborescente

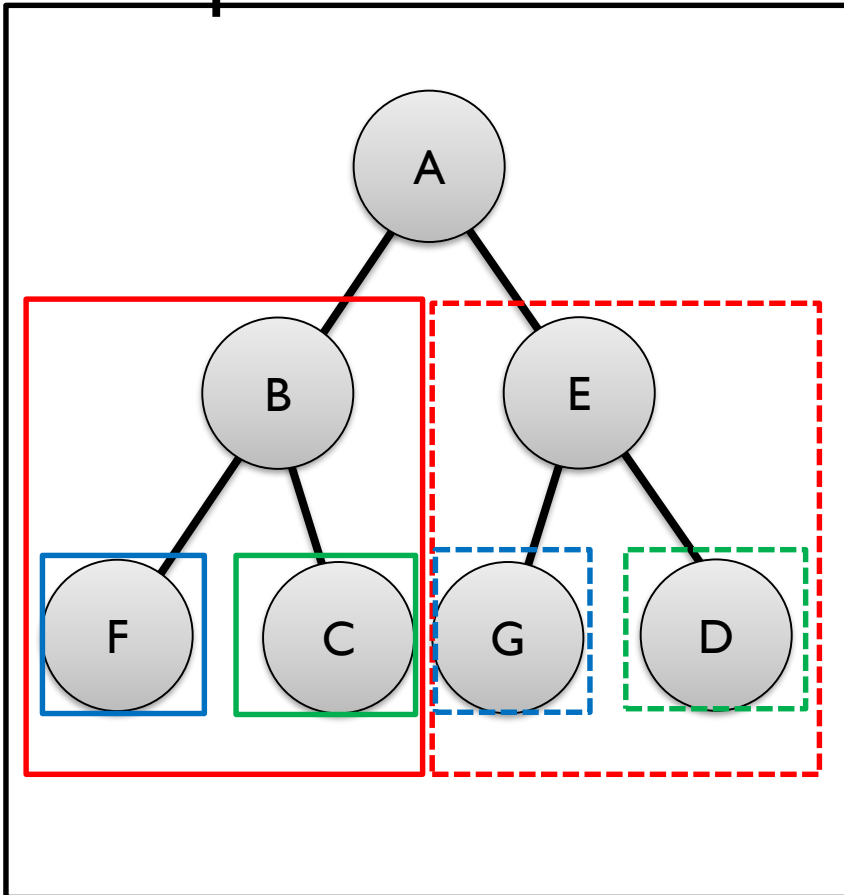
## Arbres



- Cas particulier des graphes : les arbres sont des graphes acycliques et orientés

# Structure arborescente Arbres

## Exemples



<enseignement>

<cours>

<title> Dev d'appli </title>

<niveau> H3 </niveau>

</cours>

<cours>

<title> BDD </title>

<niveau> H3 </niveau>

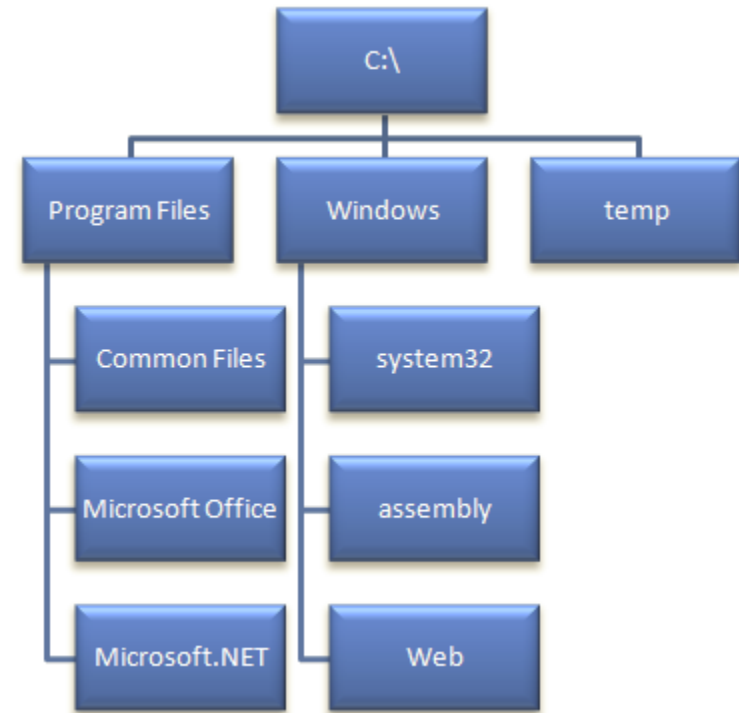
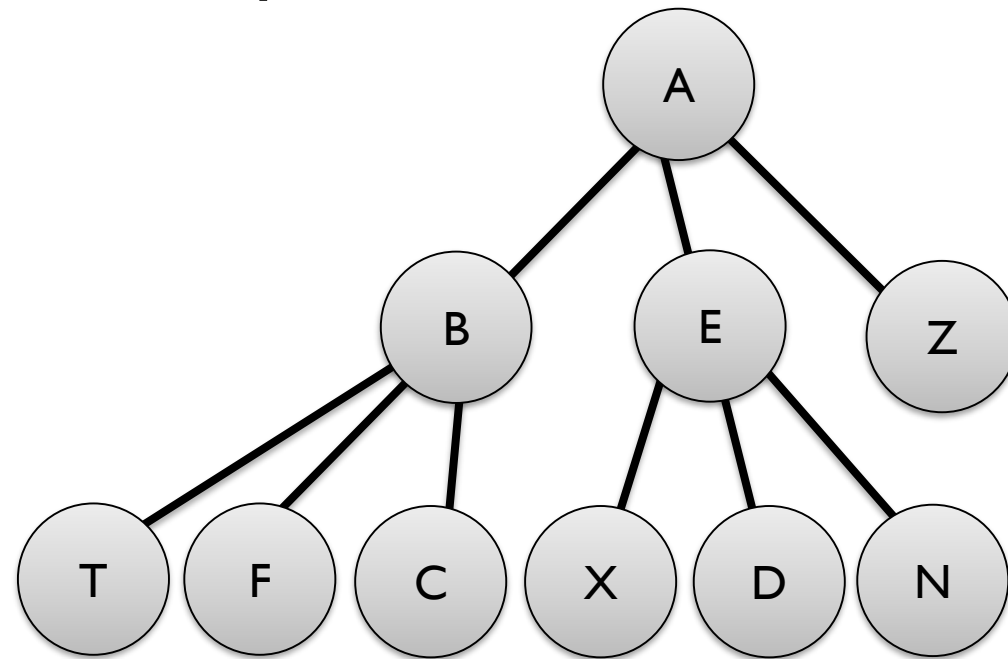
</cours>

</enseignement>

- XML

# Structure arborescente Arbres

## Exemples



- Système de fichiers

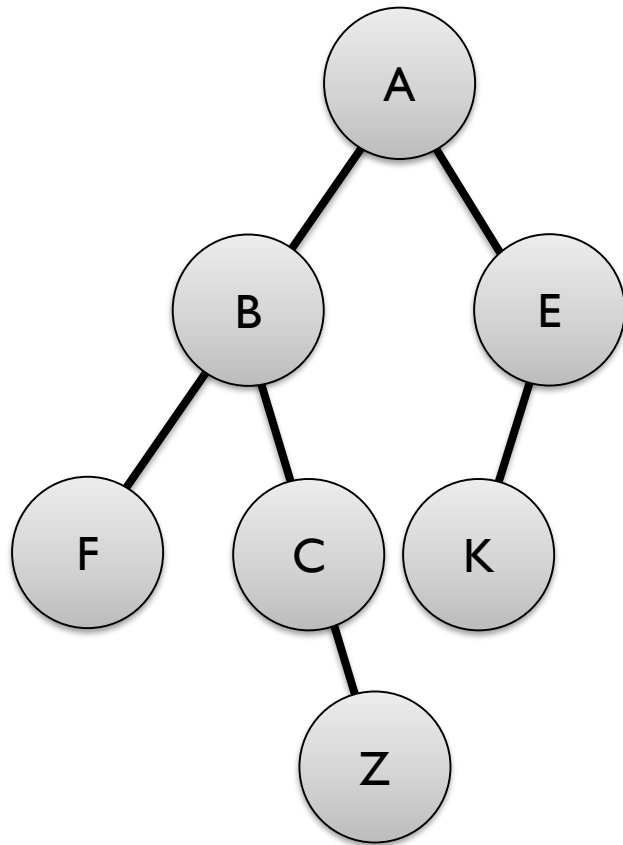
# Structure arborescente

## Arbres

- Dans nos arbres, chaque nœud peut avoir 0, 1 ou 2 successeurs.
- On parle de nœud père et de nœud(s) fils : il y a une notion d'ordre.
- Un nœud qui n'a pas de fils est appelé une feuille.
- Le nœud au sommet de la hiérarchie est la racine

# Structure arborescente

## Arbres



Arbre G

- $\text{pere}(A) = \{ \emptyset \}$
- $\text{fils}(A) = \{ B, E \}$
- $\text{pere}(B) = \text{pere}(E) = \{ A \}$
- $\text{fils}(C) = \{ Z \}$
- ....
- $\text{racine}(G) = \{ A \}$
- $\text{feuilles}(G) = \{ F, Z, K \}$

# Structure de la fenêtre JavaFX

Comme racine de l'arbre des objets graphiques, on prend l'un des objets « conteneur » proposés par JavaFX. Par exemple « **AnchorPane** ». En java, cela donne :

```
AnchorPane root = new AnchorPane();
```

On crée donc une variable locale (appelée « root ») de type « **AnchorPane** » que l'on initialise avec un nouvel objet « **AnchorPane** ».

Remarque : En programmation objet, on dit que l'on appelle le constructeur de l'objet.

# Structure de la fenêtre JavaFX

Pour faire un exemple simple on peut se contenter d'un seul objet graphique : un bouton. Il faudra donc premièrement créer le bouton. En java cela donne :

```
Button bouton= new Button("Mon Bouton");
```

On crée donc une variable locale (appelée « bouton ») de type « **Button** » que l'on initialise avec un nouvel objet « **Button** ». La chaîne de caractères passée en argument du constructeur précise le libellé du bouton : « Mon Bouton ».

Il faudra ensuite ajouter, à la liste des fils de la racine de l'arbre, le bouton.

```
root.getChildren().add(bouton);
```

La méthode `getChildren()` renvoie la liste des fils du conteneur (ici la racine `root`). La méthode `add()` ajoute à la liste l'objet graphique passé en argument.

# Structure de la fenêtre JavaFX

Puis, on crée le décor (Scene) à partir de l'arborescence préalablement construite. En Java cela donne :

```
Scene scene = new Scene(root);
```

On crée donc une variable locale (appelée « scene ») de type « **Scene** » que l'on initialise avec un nouvel objet « **Scene** ». L'argument du constructeur précise la racine de l'arborescence graphique.

Enfin, on agit sur l'estrade (Stage) en y affectant un titre via la méthode « **setTitle()** » et en y affectant le décor via la méthode « **setScene()** » et on affiche la fenêtre via la méthode « **show()** ».



# Structure de la fenêtre JavaFX

Par exemple, pour l'estrade construite par la méthode « *launch()* » et passée en paramètre (nommé « *primaryStage* ») à la méthode « *start()* » de construction de la scène, on écrira en JavaFX :

```
primaryStage.setTitle("Le titre de la fenêtre");  
primaryStage.setScene(scene);  
primaryStage.show();
```

Remarque : Quand on ferme la fenêtre principale, on sort de la boucle de gestion des événements de l'IHM et de la méthode « *launch()* ».

# Programmation de GUI avec JavaFx

On obtient donc le code suivant :

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Stage;

public class MainApps extends Application {
    public void start(Stage primaryStage) throws Exception {
        AnchorPane root = new AnchorPane();

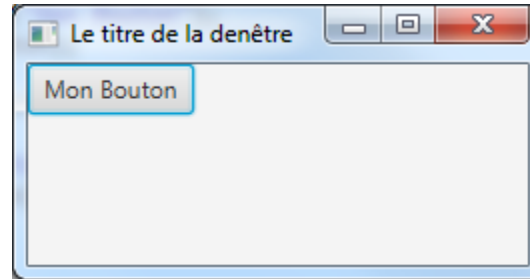
        Button bouton= new Button("Mon Bouton");
        root.getChildren().add(bouton);

        Scene scene = new Scene(root);

        primaryStage.setTitle("Le titre de la fenêtre");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

# Programmation de GUI avec JavaFx



On peut cliquer sur le bouton, rien ne se passe. En fait, l'IHM gère les événements mais il n'y a pas d'algorithme écrit pour l'instant pour traiter ces événements. Donc aucune action n'est déclenché.

On peut seulement redimensionner la fenêtre ou la fermer

# TD

- Modifiez le code de l'application JavaFX que l'on vient de construire de façon à y ajouter un nouveau bouton « Bouton 2 » dans la fenêtre(en plus du bouton déjà crée).
- Appelez les méthodes « **setLayoutX(double value)** » et « **setLayoutY(double value)** » sur le second bouton afin que les deux boutons ne se chevauchent plus. (Remarque : sans cela les deux boutons apparaissent aux mêmes coordonnées (0,0) et donc se superposent.) pour cela:
  - Créer le bouton 2
  - bouton2.setLayoutX(100)
  - bouton2.setLayoutY(0)
- Dessinez l'arbre des objets graphiques utilisés par la « Scene »

Pour chacun des TD veuillez créer un nouveau package avec les fichiers utiles à l'exécution du projet

# Plan du cours

- Introduction
- Programmation Java
- **Le FXML**
- Utilisation d'un RAD
- Disposition adaptative
- Traitement des événements
- Application multifenêtres
- Annexes

# Description fxml d'une GUI

Au lieu de programmer en Java l'imbrication des différents objets graphiques composant l'arbre de notre scène, il est possible d'utiliser un langage de description de l'arbre.

Ce langage s'appelle FXML et est un dérivé du langage XML., créé spécialement pour JavaFX,

# Description fxml d'une GUI

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.control.Button?>
```

```
<?import javafx.scene.layout.AnchorPane?>
```

```
<AnchorPane xmlns="http://javafx.com/javafx/8.0.111"  
xmlns:fx="http://javafx.com/fxml/1">
```

```
  <children>
```

```
    <Button text="Button" />
```

```
  </children>
```

```
</AnchorPane>
```

# Le langage XML

C'est un langage de balise (ou tag en anglais). Une balise `<NOM>` définit le début d'une section qui se termine par la balise de fermeture `</NOM>`. Si une section est vide, on peut regrouper la balise ouvrante et la balise fermante sous une seule balise `<NOM />`.

La balise ouvrante (ou la balise seule) peut contenir des attributs qui sont alors indiqués par la syntaxe : `<NOM ATTRIBUT1="VALEUR1" ATTRIBUT2="VALEUR2">`.

La première ligne du fichier contenant la balise `<?xml version="1.0" encoding="UTF-8"?>` permet d'indiquer que c'est un fichier XML et le type d'encodage du jeu de caractères (ici le standard UTF8).

Les balises `<?import ...?>` permettent d'inclure des descriptions XML extérieures au fichier.

La balise `<!-- commentaire -->` permet d'inclure un commentaire.

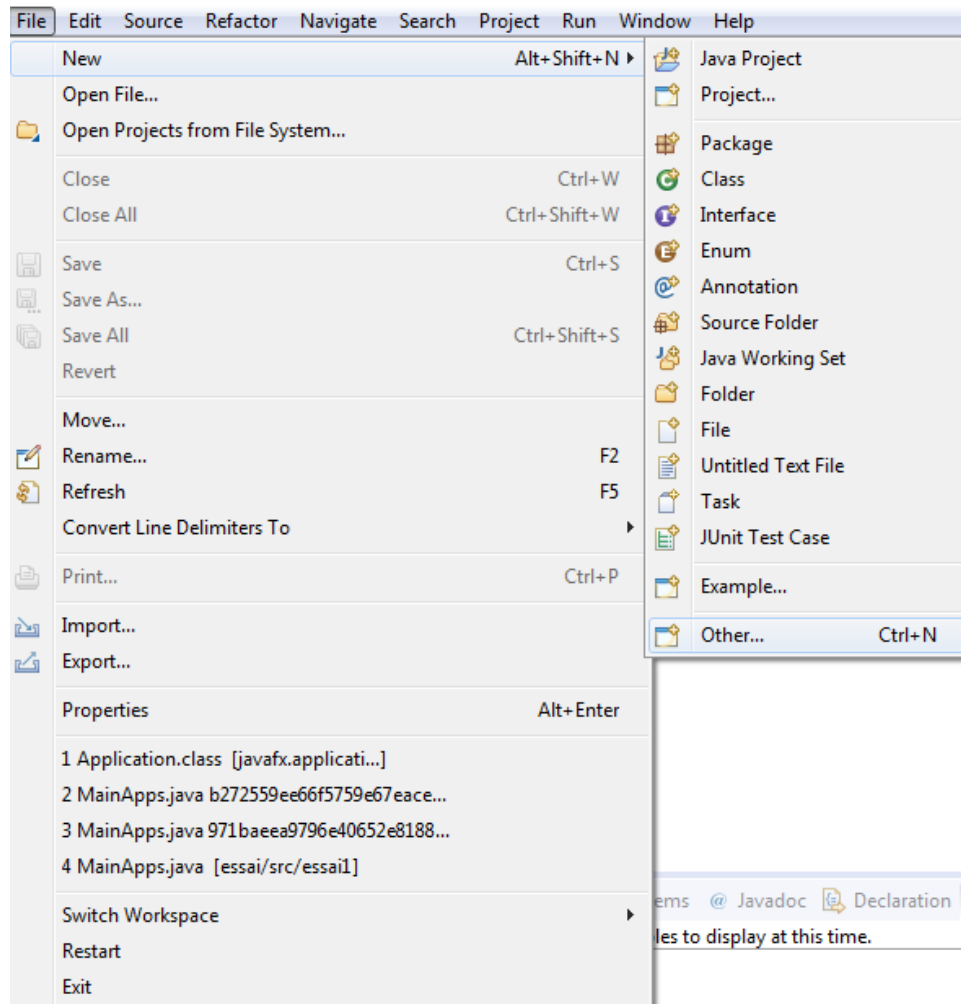


# Le langage XML

Les zones définies via les balises peuvent s'imbriquer. On forme ainsi l'arbre XML en partant d'une balise racine. La zone définie par la racine contiendra les balises de ses nœuds fils.

Il existe différentes utilisations du langage XML, par exemple XHTML en est une. FXML en est une autre. La syntaxe XML est toujours la même, mais la grammaire (c'est-à-dire les noms des balises, de leurs attributs et la façon dont on est autorisé à les agencer dans l'arbre XML) diffère.

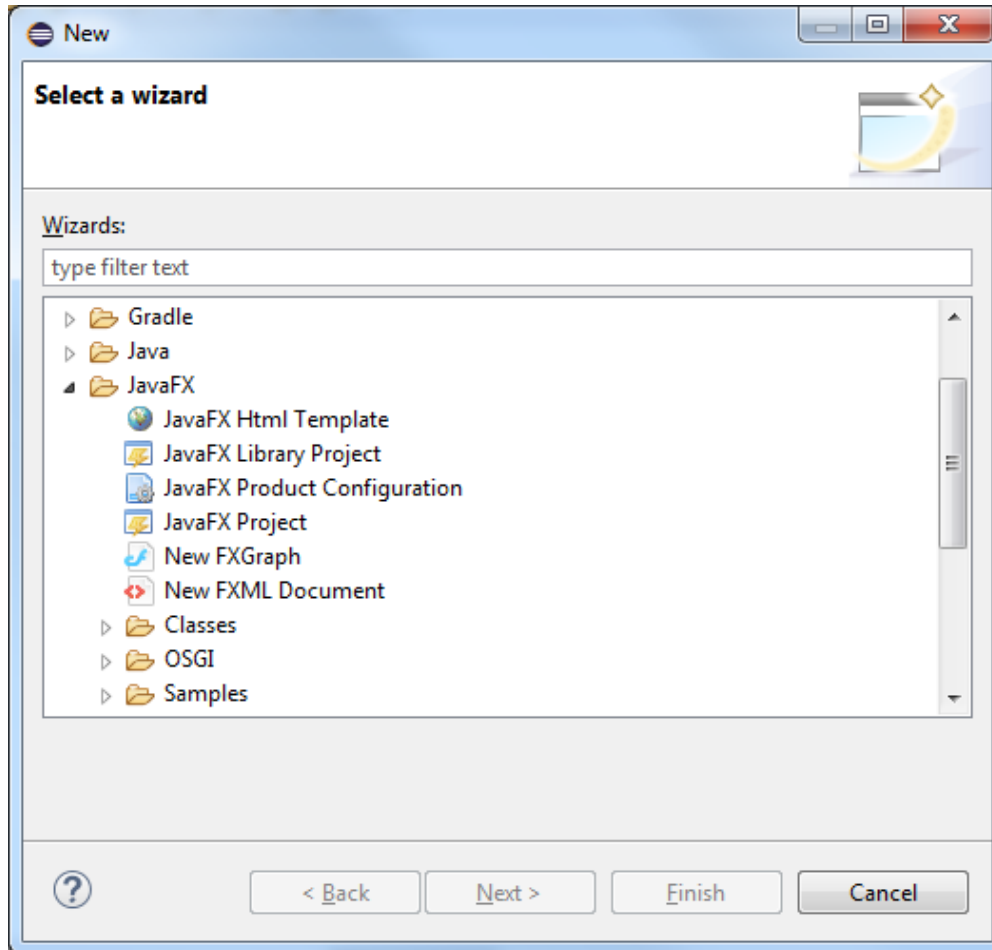
# Création du fichier dans eclipse



**Si vous avez opté la 1ère méthode pour créer votre projet (Projet Java normal), il faut bien sùre créer un fichier fxml sinon un fichier fxml de base existe déjà dans votre projet.**

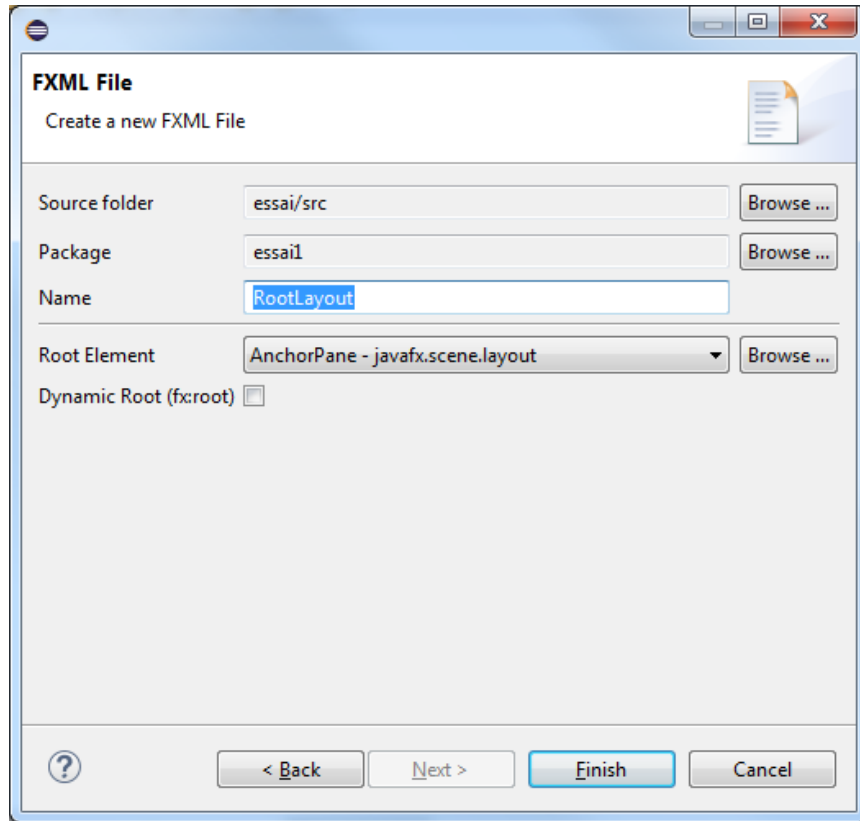
Dans le menu « File » et son sous-menu « New », on sélectionne l'item « Other »

# Création du fichier dans eclipse



On ouvre la liste « JavaFX » et on sélectionne « New FXML Document »

# Création du fichier dans eclipse



On indique le nom du fichier que l'on souhaite créer.

Par exemple : « RootLayout »

Puis on clique sur le bouton « Finish ».

Un fichier « RootLayout.fxml » sera ainsi créé contenant :

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.AnchorPane?>

<AnchorPane xmlns:fx="http://javafx.com/fxml/1">
    <!-- TODO Add Nodes -->
</AnchorPane>
```

On pourra ensuite l'éditer et le compléter pour obtenir le contenu du transparent n°31

# Chargement d'une GUI fxml par JavaFx

On va écrire du code JavaFX pour charger le fichier fxml et construire en conséquence l'arborescence des objets graphiques. Pour cela, il faut faire appel à un objet de JavaFX qui va faire le travail de chargement. Il s'appelle « **FXMLLoader** ».

On va donc déclarer une variable locale de type « **FXMLLoader** » que l'on initialisera avec un objet « **FXMLLoader** ». En JavaFX, cela donne par exemple :

```
FXMLLoader loader = new FXMLLoader();
```

Puis, on indique le fichier fxml (en recherchant son chemin) via la méthode « **setLocation()** » du loader.

```
loader.setLocation(MainApp.class.getResource("RootLayout.fxml"));
```

La méthode « **MainApp.class.getResource()** » transforme le nom de fichier en URL.

# Chargement d'une GUI fxml par JavaFx

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Stage;
```

```
public class MainApps extends Application {
```

```
    public void start(Stage primaryStage) throws Exception{
```

```
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(MainApps.class.getResource("RootLayout.fxml"));
        AnchorPane rootLayout = (AnchorPane) loader.load();
```

```
        Scene scene = new Scene(rootLayout);
        primaryStage.setTitle("Le titre de la fenêtre");
        primaryStage.setScene(scene);
        primaryStage.show();
```

```
    }
```

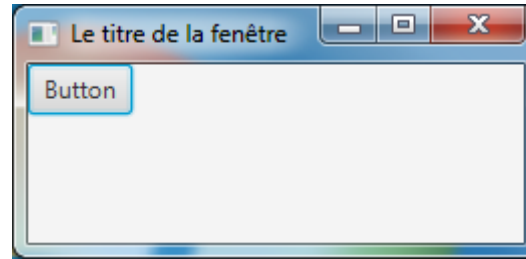
```
    public static void main(String[] args) {
```

```
        launch(args);
```

```
    }
```

```
}
```

# Chargement d'une GUI fxml par JavaFx



On obtient le même résultat que par la programmation directe en Java.

Mais on n'a pas à gérer des noms de variables locales pour chaque objet graphique que l'on crée ni à appeler des méthodes. Il faut seulement connaître le nom des Objets graphiques et leurs attributs.

# TD

- Modifiez le code FXML de l'application, que l'on vient de construire, de façon à placer deux boutons « Bouton 1 » et « Bouton 2 » dans la fenêtre.
- En plaçant les attributs XML « **layoutX** » et « **layoutY** » sur le second bouton, faire en sorte que les deux boutons ne se chevauchent plus. (Remarque : sans cela les deux boutons apparaissent aux mêmes coordonnées (0,0) et donc se superposent.)
- Comparez l'arbre XML et l'arbre dessiné au TD précédent.



# Plan du cours

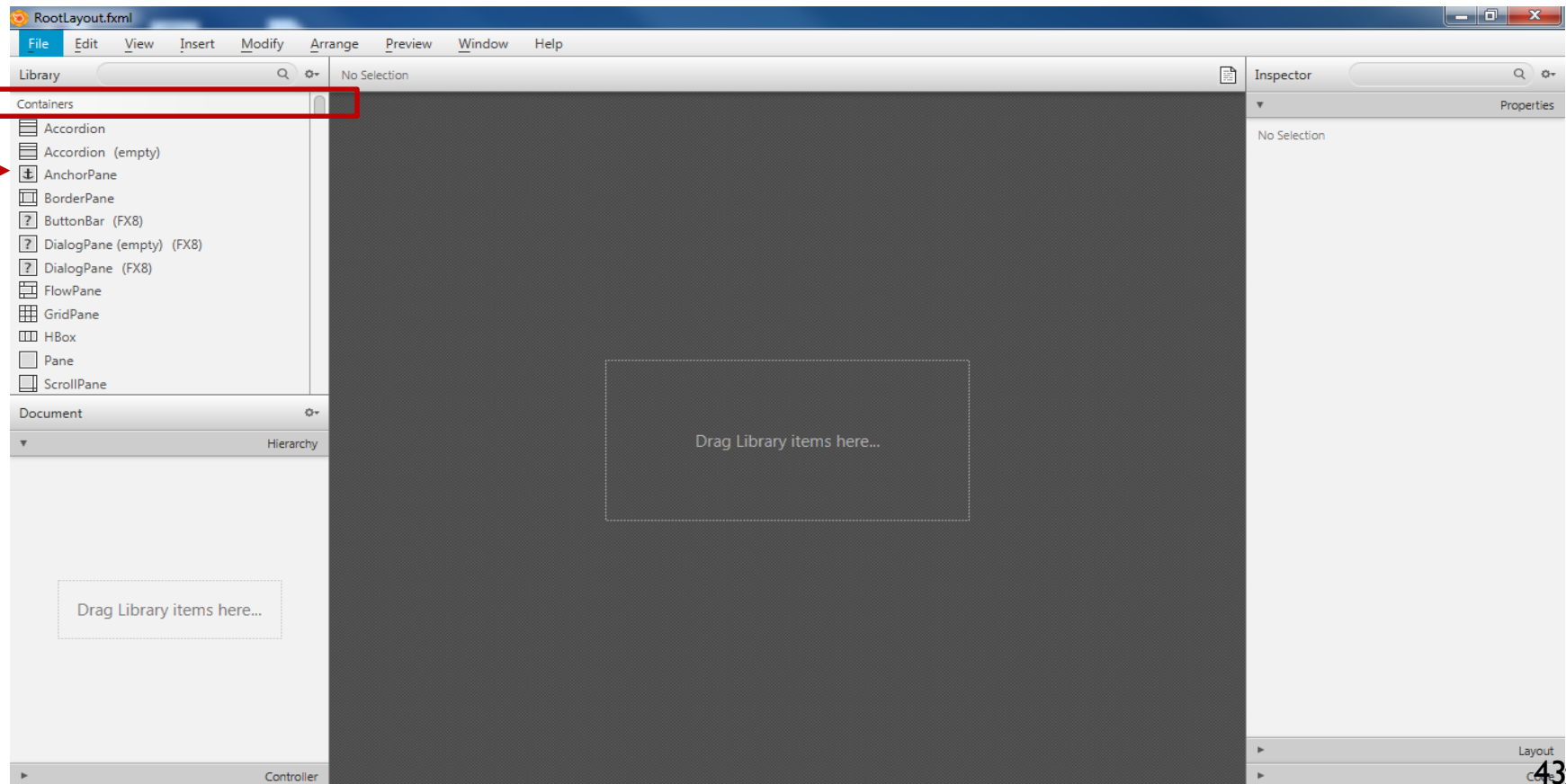
- Introduction
- Programmation Java
- Le FXML
- **Utilisation d'un RAD**
- Disposition adaptative
- Traitement des événements
- Application multifenêtres
- Annexes

# Construction du fxml avec SceneBuilder

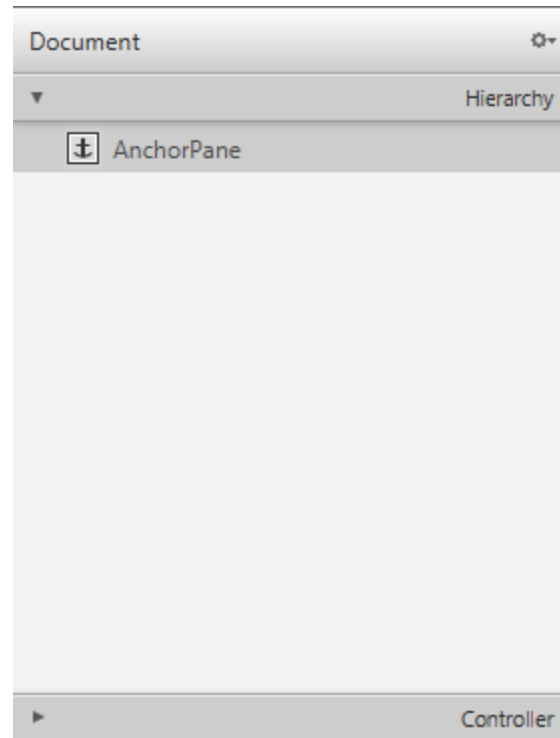
Au lieu d'écrire directement du FXML, on va utiliser le RAD (Rapide Application Designer) SceneBuilder afin de construire ou modifier des fichiers FXML.

# Construction du fxml avec SceneBuilder

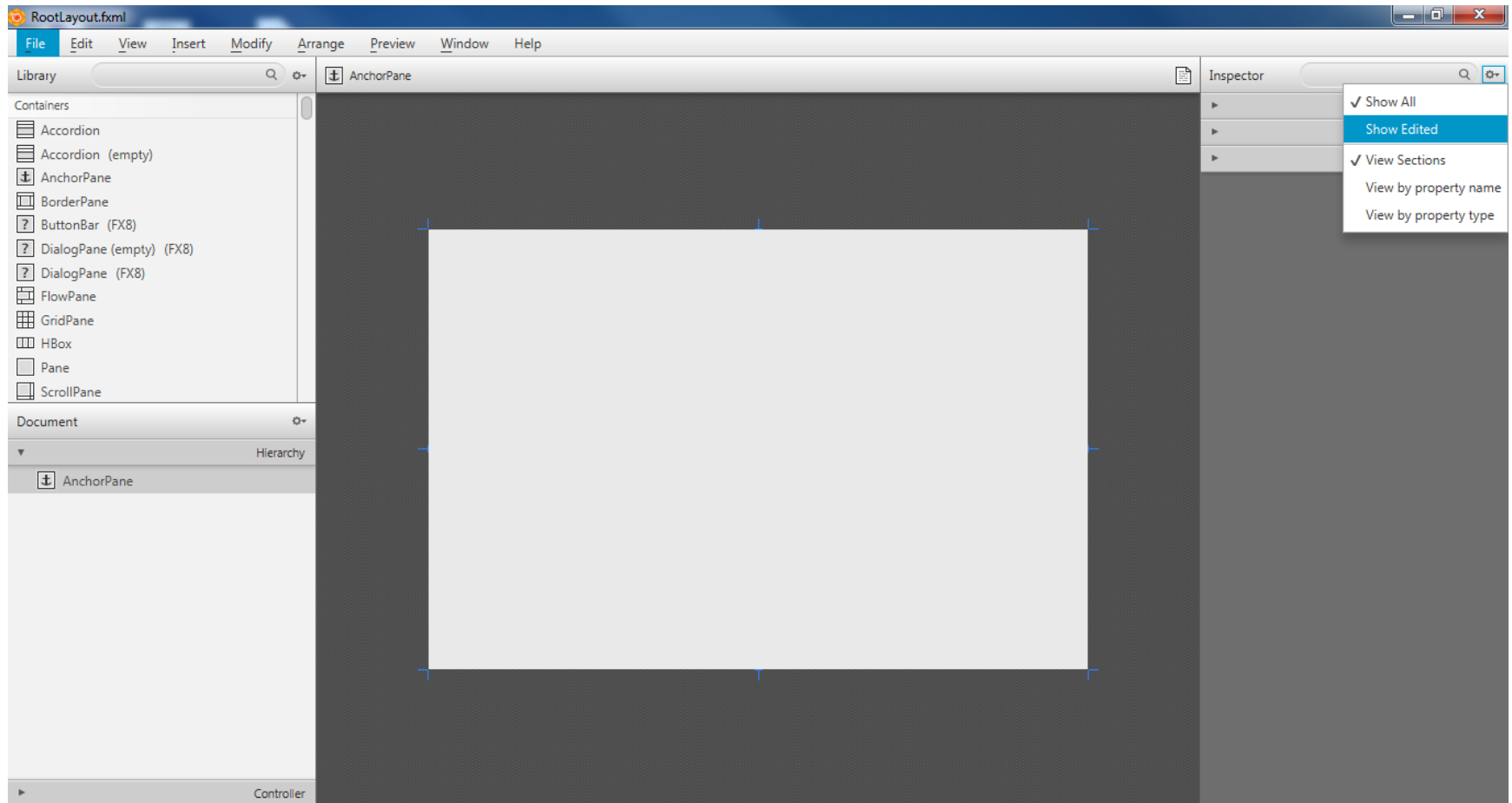
Pour ouvrir cet éditeur, faire un clic droit sur le fichier fxml que vous venez de créer puis « Ouvrir avec SceneBuilder »



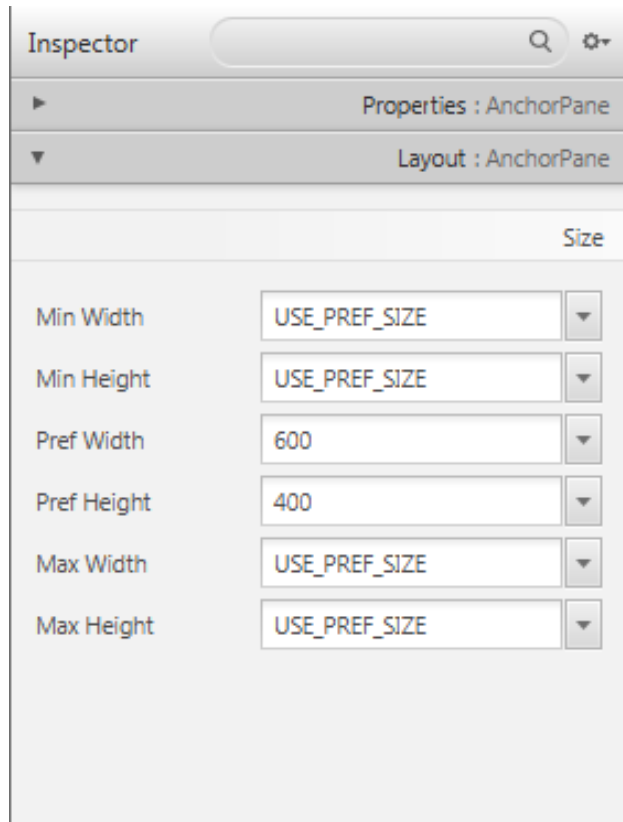
# Construction du fxml avec SceneBuilder



# Construction du fxml avec SceneBuilder



# Construction du fxml avec SceneBuilder



Ayant choisit via l'option « Show Edited » de ne visualiser que les attributs dont la valeur n'est pas celle par défaut, on voit seulement 6 attributs dans la rubrique « Layout ». Ce sont ceux que SceneBuilder avait déjà pré-modifiés pour nous.

En cliquant sur une petite roue apparaissant à droite de la flèche ou en sélectionnant la valeur par défaut via la flèche, on va les faire disparaître une à une pour être conforme à ce que l'on a programmé directement en JavaFx.

L'AnchorPane apparaît alors réduite à un pixel que l'on ne voit pas.

# Construction du fxml avec SceneBuilder

- Dans l'AnchorPane, on fait glisser l'objet graphique « button » que l'on trouve dans la sous-liste « Controls ».
- Les dimensions de visualisation de l'AnchorPane s'adapte à son contenu et donc aux dimensions du « button ».
- En sauvegardant le fichier (dans le projet Java), on obtient le même contenu pour le fichier fxml.
- Pour voir le résultat, aller sur l'onglet « preview » puis « show preview in window » pour voir le résultat

# TD

- Utilisez SceneBuilder pour modifier l'application, que l'on vient de construire, de façon à placer deux boutons « Bouton 1 » et « Bouton 2 » dans la fenêtre comme on l'avait fait au précédent TD en agissant directement dans le fichier FXML.



# Plan du cours

- Introduction
- Programmation Java
- Le FXML
- Utilisation d'un RAD
- **Disposition adaptative**
- Traitement des événements
- Application multifenêtres
- Annexes

# Amélioration du rendu Via SceneBuilder

En jouant uniquement sur les attributs « layoutX » et « layoutY » d'objet graphique situé dans le conteneur racine, on va seulement pouvoir construire des applications ayant un rendu figé indépendant de la taille de la fenêtre.

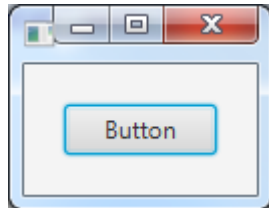
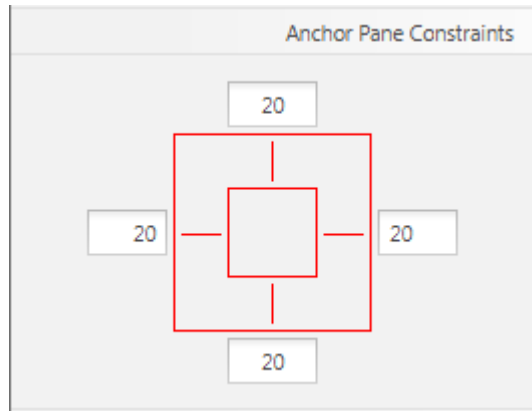
JavaFX propose un certain nombre de conteneurs ayant des capacités de redimensionnement ou de positionnement automatique d'objet graphique.

# Amélioration du rendu Via SceneBuilder

En sélectionnant le « button » (soit au niveau graphique, soit depuis la hiérarchie), on peut accéder à tous les attributs de l'objet via les rubriques « Properties » (pour tout ce qui définit l'objet) et « Layout » (pour tout ce qui le lie à sa boîte englobante).

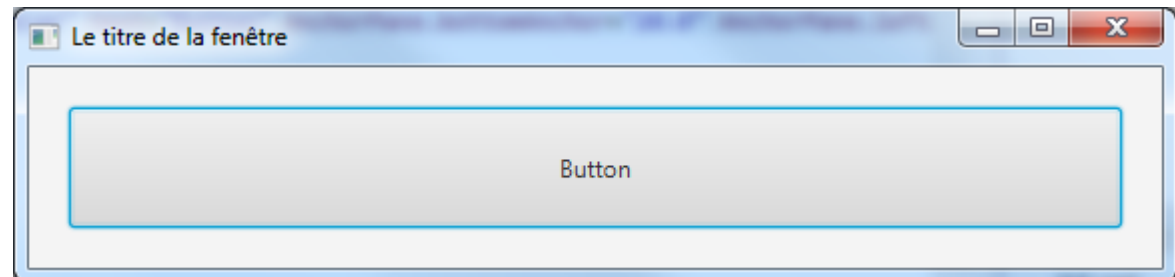
Via « **Layout** », on pourra notamment accéder aux dimensions externes.

# Amélioration du rendu Via SceneBuilder



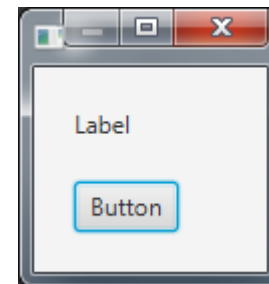
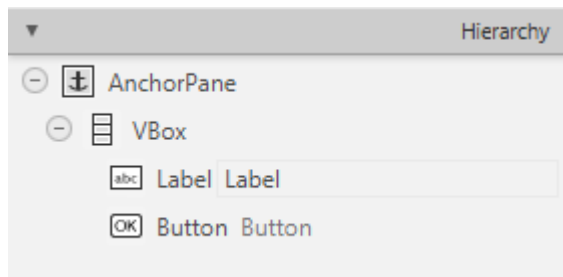
Via les attributs « **Anchor Pane Constraints** », on peut lier le bord droit de l'objet à une certaine distance (20 pixels dans l'exemple ci contre) du bord droit du conteneur. Idem, pour les trois autres bords. On lie ainsi les dimensions du conteneur aux dimensions de l'objet.

Au lancement de l'application, on obtient la fenêtre ci-contre, et si on redimensionne la fenêtre, on obtient par exemple la fenêtre ci-dessous.

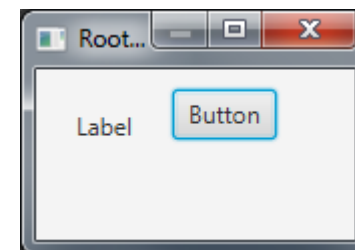
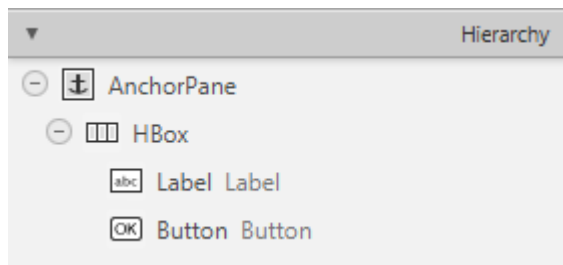


# Les VBox et HBox

Le conteneur « **VBox** » va permettre d'aligner plusieurs objets (ou plusieurs hiérarchies d'objets) verticalement, c'est-à-dire les uns en dessous des autres.

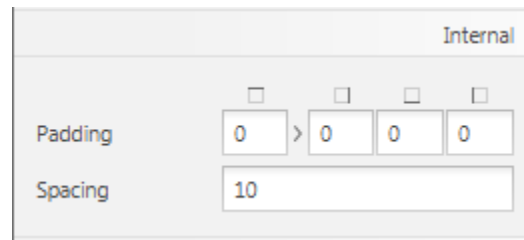


Le conteneur « **HBox** » va permettre d'aligner plusieurs objets (ou plusieurs hiérarchies d'objets) horizontalement, c'est-à-dire les uns à coté des autres.



# Les VBox et HBox

Les conteneurs « VBox » « Hbox » disposent d'un attribut disponible dans « SceneBuilder » à la rubrique « **Layout / Internal** ». Il est nommé « **Spacing** » et définit le nombre de pixel de séparation entre chaque enfant.

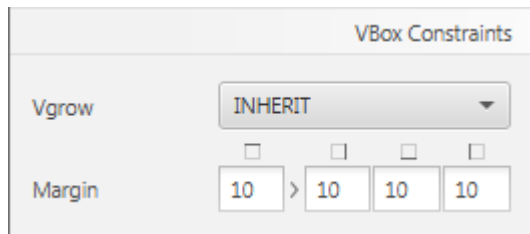


Dans l'exemple, on définit ainsi qu'il y aura un espace de 10 pixels entre le label et le bouton.

# Les VBox et HBox

Le conteneur « VBox » **offre** à ces objets ou conteneurs **enfants** un attribut de positionnement : « **Vgrow** » ainsi qu'un attribut de marge.

Dans l'enfant, on peut donc indiquer s'il doit (valeur « ALWAYS ») ou ne doit pas (valeur « NEVER ») adapter sa hauteur (dimension verticale) en fonction de la hauteur globale du « VBox » et indiquer les marges que l'on souhaite.



VBox Constraints

Vgrow: INHERIT

Margin: 10 > 10 10 10



HBox Constraints

Hgrow: INHERIT

Margin: 0 > 0 0 0

De même, le conteneur « HBox » offre à ces objets ou conteneurs enfants un attribut de positionnement : « **Hgrow** ».

Dans l'enfant, on peut donc indiquer s'il doit (valeur « ALWAYS ») ou ne doit pas (valeur « NEVER ») adapter sa largeur (dimension horizontale) en fonction de la largeur globale du « Hbox ».

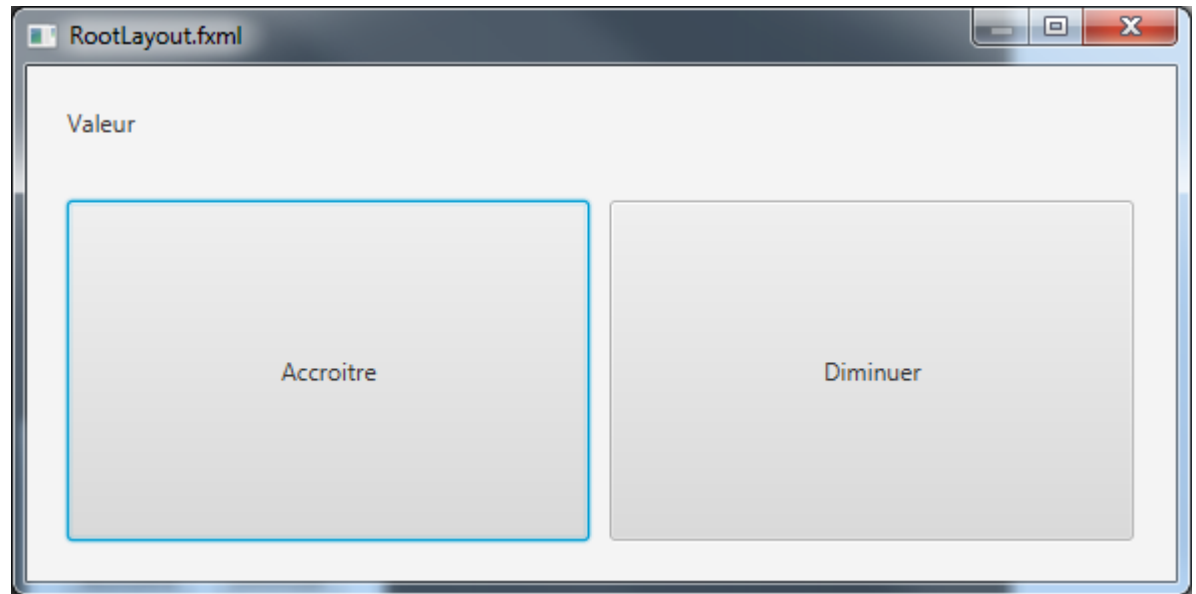
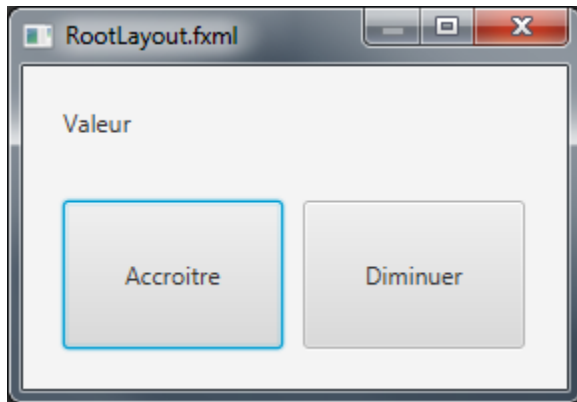
# Le conteneur « AnchorPane »

Comme on a commencé à le voir dans les exemples précédents le conteneur « AnchorPane » permet de positionner un objet ou un autre conteneur en lui permettant le redimensionnement automatique via les attributs de contrainte.

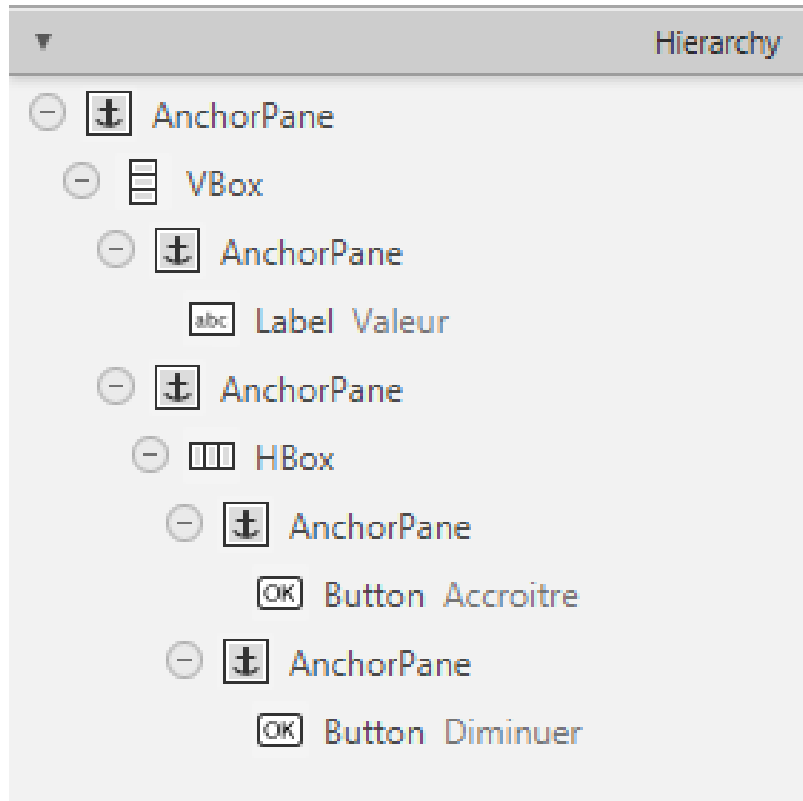


# Hierarchie de conteneur

A titre d'exemple, on va essayer de construire l'interface suivante, dans laquelle les boutons « Accroître » et « Diminuer » vont voir leur taille s'adapter à la taille de la fenêtre alors que la hauteur occupée par le label ne variera pas.



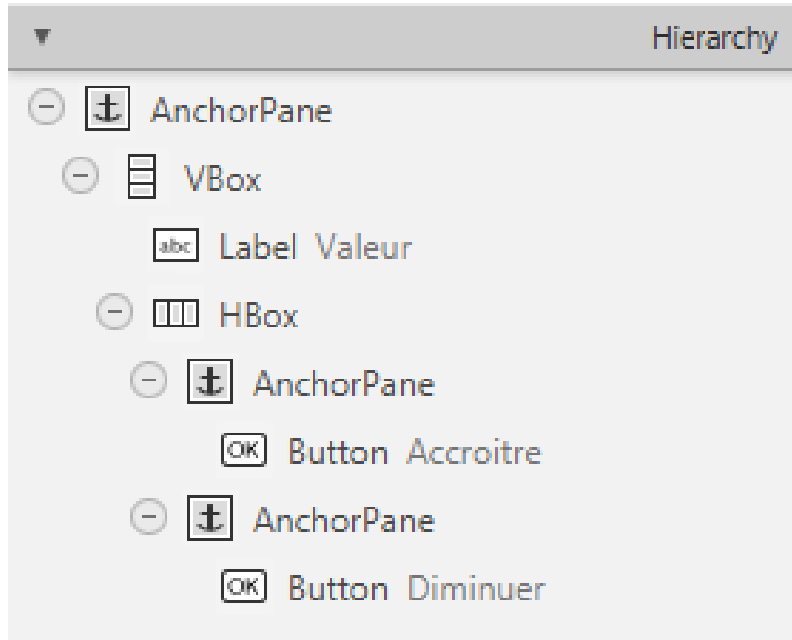
# Hierarchie de conteneur



Comme la méthode « start » définit la racine comme un « AnchorPane », on doit respecter cela dans notre hiérarchie de conteneur.

D'autre part : faire précéder chaque conteneur, ou objet graphique, par un « AnchorPane » permettra d'adapter la taille de l'élément à l'espace offert par le conteneur précédent.

# Hierarchie de conteneur

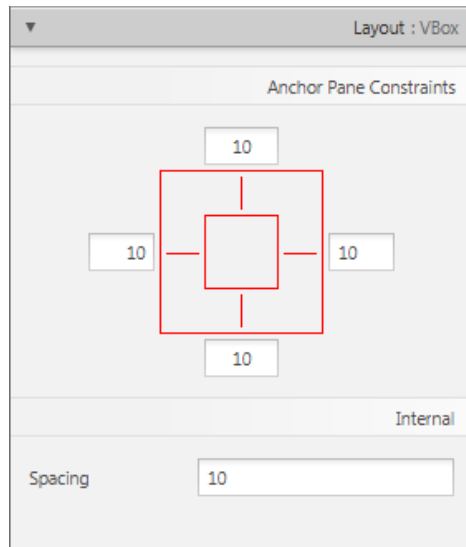


Il est également possible de mettre le Label directement dans la « VBox » sans « AnchorPane » intermédiaire, car le texte du « Label » ne peut s'étendre

De même, on pourra mettre le « HBox » directement dans le « VBox ».

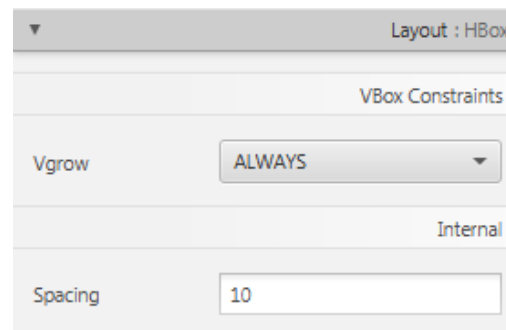
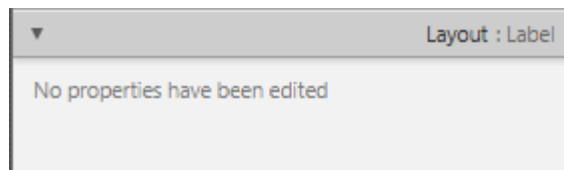
On obtient alors une hiérarchie un peu simplifiée comme celle ci-contre.

# Hierarchie de conteneur



En dessous de la racine, à chaque niveau de l'arbre, il convient de fixer les attributs hérités du conteneur qui le précède.

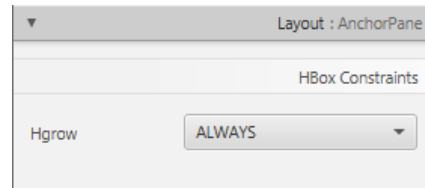
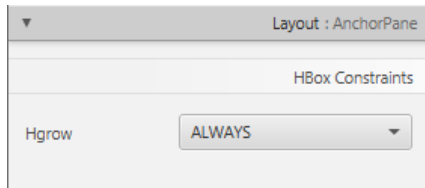
La « VBox » doit occuper tout le volume (à une marge prêt 10 pixels par exemple)



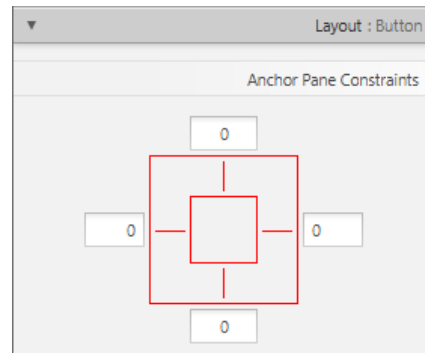
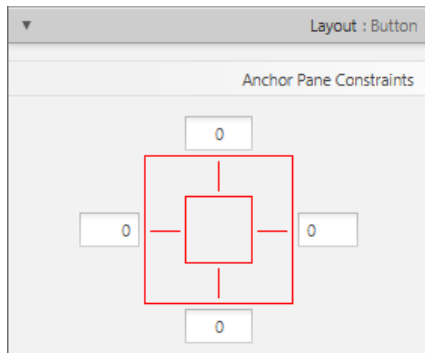
Le « Label » garde ses valeurs par défaut, tandis que l'on positionne à « ALWAYS » l'attribut « Vgrow » de la « HBox »

# Hierarchie de conteneur

De la « Hbox », partent deux branches de configuration identique :



On a d'abord une « AnchorPane » dont on positionne à « ALWAYS » l'attribut « Hgrow ».



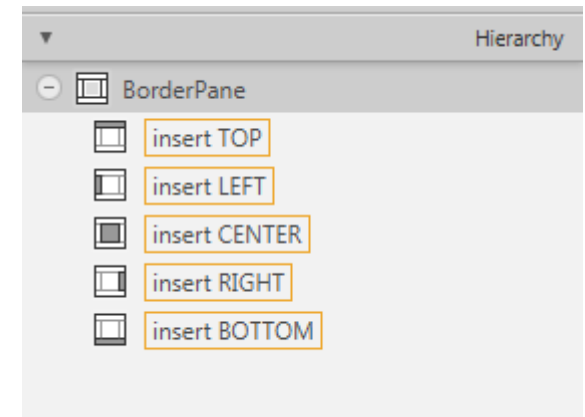
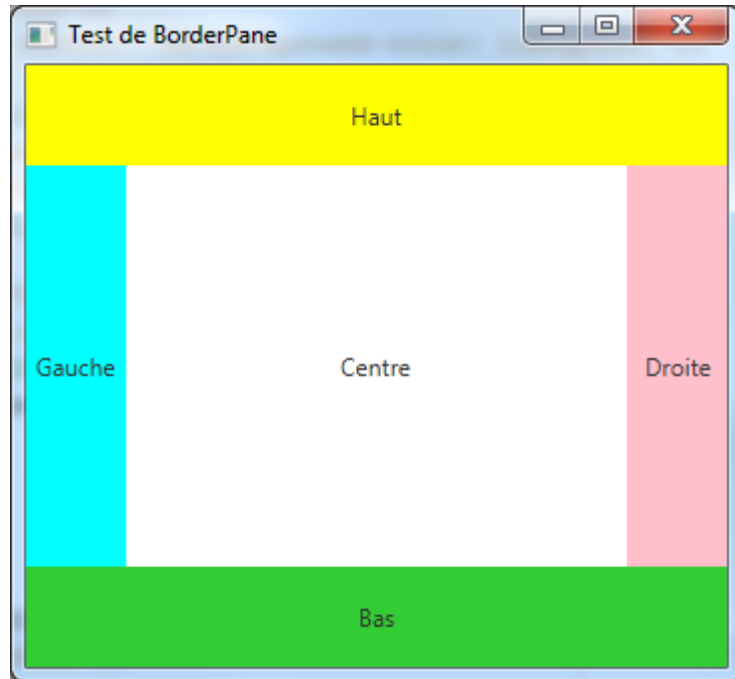
Ensuite on a le « Button » qui occupera tout l'espace de son conteneur.

# D'autres conteneurs

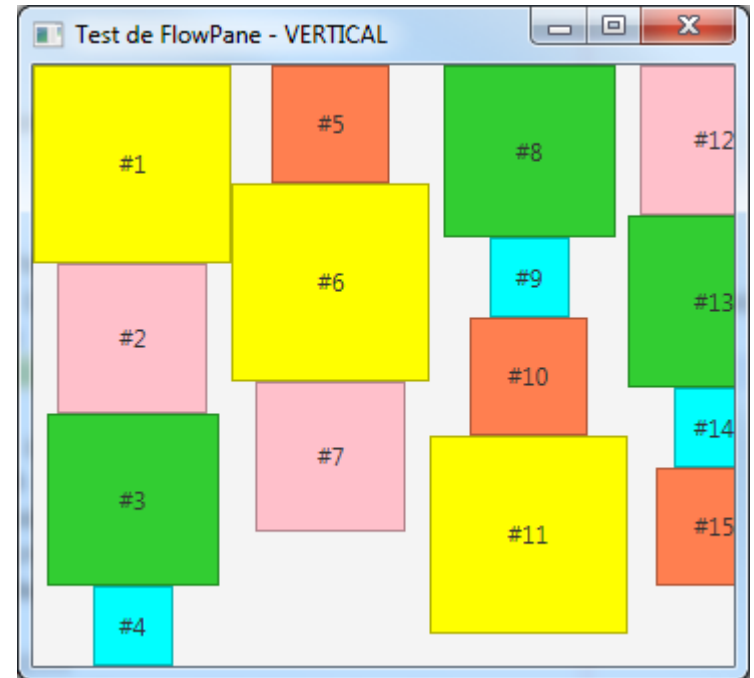
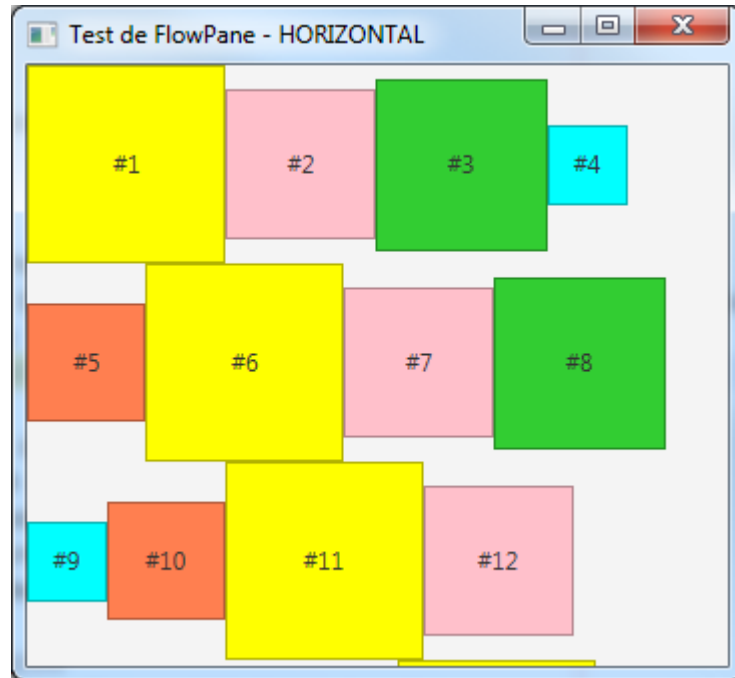
Il existe également d'autres conteneurs autres que les « AnchorPane », « VBox » et « HBox » :

- BorderPane
- FlowPane
- TilePane
- GridPane
- ScrollPane
- Accordion

# BorderPane



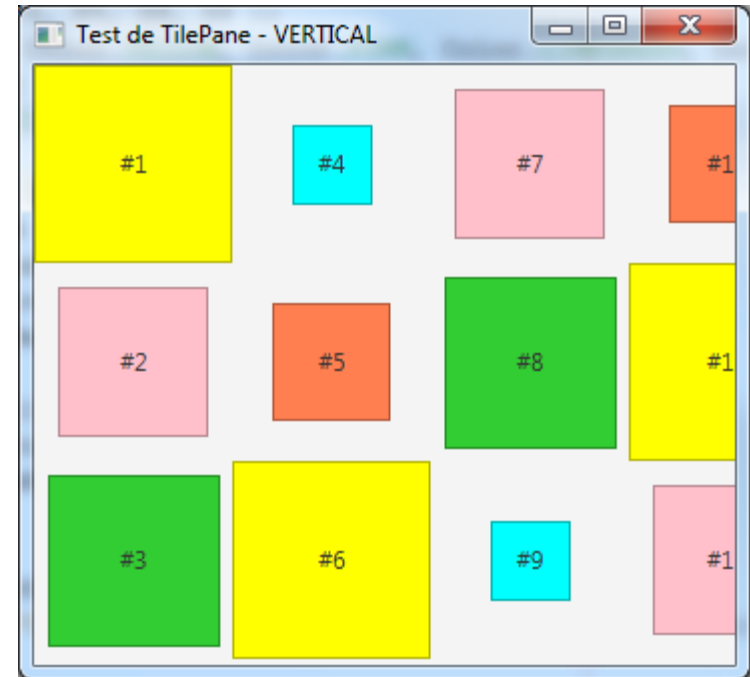
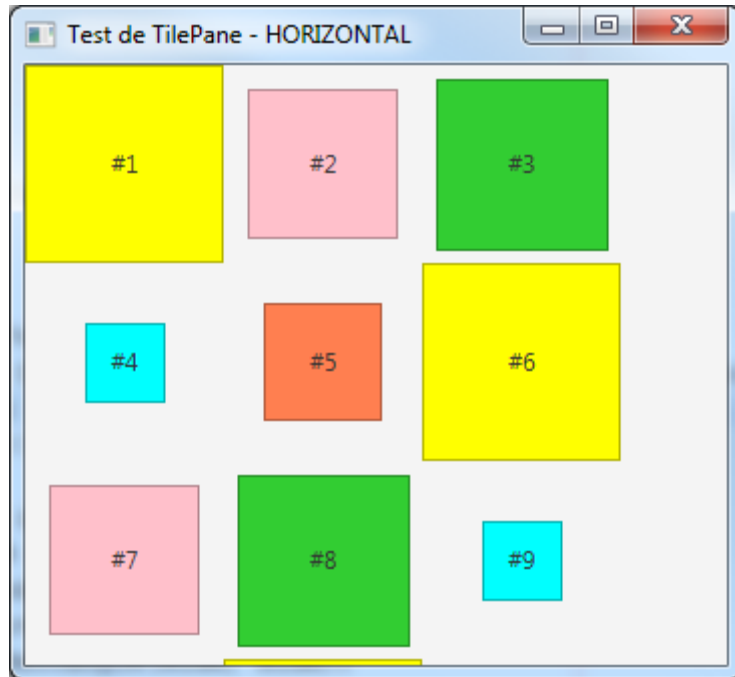
# FlowPane



En fonction de la valeur de l'attribut : « Orientation » qui peut prendre les valeurs « HORIZONTAL » ou « VERTICAL ».



# TilePane



En fonction de la valeur de l'attribut : « Orientation » qui peut prendre les valeurs « HORIZONTAL » ou « VERTICAL ».

# Ce qu'il faut retenir

- Il faut toujours architecturer son application avant de commencer à la développer
  - ✓ Quels sont les différents composants qui vont constituer votre applications
  - ✓ Quels sont les conteneurs nécessaires
  - ✓ Associer les composants aux conteneurs

# TD

- En utilisant SceneBuilder, construire une fenêtre ayant un menu en haut et un quadrillage de 2 images sur 2 images, chaque image possédant des scrolls pour se déplacer dans l'image quand la fenêtre de visualisation est plus petite que l'image.



- En utilisant SceneBuilder, construire une fenêtre ayant un menu en haut et un quadrillage de 2 images sur 2 images, chaque image possédant des scrolls pour se déplacer dans l'image quand la fenêtre de visualisation est plus petite que l'image.

Remarque : On pourra utiliser l'objet graphique (de type « controls ») « MenuBar » pour fabriquer la bar de menu. Et pour faire apparaître une image (sans les scrolls) via l'objet graphique (de type « controls ») « ImageView ».

Plutôt que d'utiliser une cascade de « VBox » et « Hbox », nous vous proposons d'utiliser un « BorderPane » et de découvrir par vous-même le « GridPane ».

Enfin, plutôt que d'utiliser les objets de control « ScrollBar » qui nécessiteraient de la programmation pour les gérer, on pourra avantageusement utiliser le conteneur « ScrollPane » (dont on pourra activer la propriété « Pannable »).

# Inclusion de FXML

Il est possible de séparer dans plusieurs fichiers la description FXML d'une fenêtre.

Prenons le cas, par exemple, d'une fenêtre utilisant, dans l'arborescence des conteneurs, un « TabPane » contenant plusieurs « Tab ».

Pour chaque « Tab », on construit le fichier FXML de sa sous-arborescence, permettant ainsi de déléguer plus facilement la conception de cette partie de l'interface.

# Inclusion de FXML

Ensuite dans le fichier FXML racine, on ajoute la balise « `<fx:include source="fichier.fxml" />` » dans le nœud qui doit procéder à l'inclusion du sous arbre défini par « `fichier.fxml` ».

L'inclusion peut également être réaliser via le RAD « SceneBuilder » en allant dans le menu « file » sous menu « include » item « FXML » après avoir sélectionné le nœud concerné.

# Plan du cours

- Introduction
- Programmation Java
- Le FXML
- Utilisation d'un RAD
- Disposition adaptative
- **Traitement des événements**
- Application multifenêtres
- Annexes



# Contrôler les événements en JAVA

Pour programmer les algorithmes de traitement des événements de l'IHM, il va falloir créer un second fichier JAVA définissant une nouvelle classe que l'on appellera « **Controller** ».

Cette **classe ne contiendra pas** de méthode « `public static void main(String[] )` », car elle n'a pas à contenir un point de lancement du programme à la différence de « `MainApp` ».



# Contrôler les événements en JAVA

La classe « vide » doit contenir :

- D'une part, sa méthode de construction. (C'est une méthode définit en programmation objet comme portant le même nom que la classe devant être public et retournant l'objet construit.).
- D'autre part, une méthode « initialize() » qui sera appelée par le système de chargement du fichier fxm1 après avoir appelé le constructeur. Elle doit être préfixée par « **@FXML** ».

# Contrôler les événements en JAVA

On commence donc par avoir le code suivant :

```
import javafx.fxml.FXML;
```

```
public class Controller {
```

```
    public Controller() {  
    }
```

```
    @FXML
```

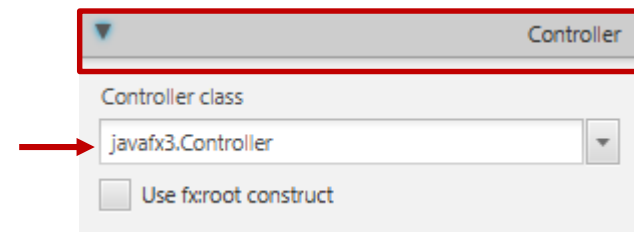
```
    private void initialize() {  
    }
```

```
}
```

# Contrôler les événements en JAVA

Dans le fichier fxml, on va devoir préciser le chemin du fichier servant de contrôleur d'événements en ajoutant dans la balise racine l'attribut : « **fx:controller="chemin"** ». La valeur « chemin » est à remplacer par le nom du package, point et le nom de la classe.

C'est bien sûr faisable via le RAD (Rapide Application Développement) SceneBuilder.



# Contrôler les événements en JAVA

Pour chaque action, on va créer une méthode (préfixée par `@FXML`) qui contiendra le code Java correspondant à l'algorithme de traitement de l'action.

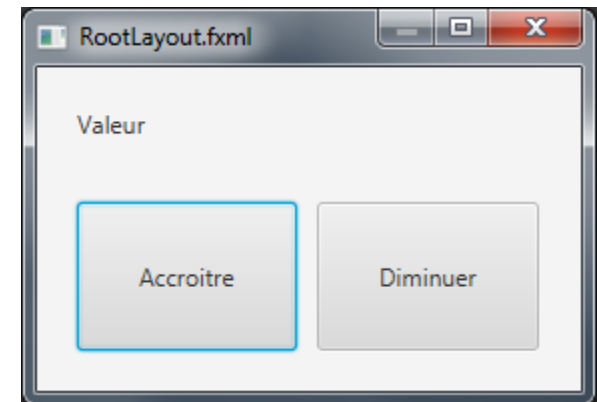
Pour la gestion des deux boutons de l'exemple, on ajoutera dans la classe « controller »

`@FXML`

```
private void handleAccroitre(){  
}
```

`@FXML`

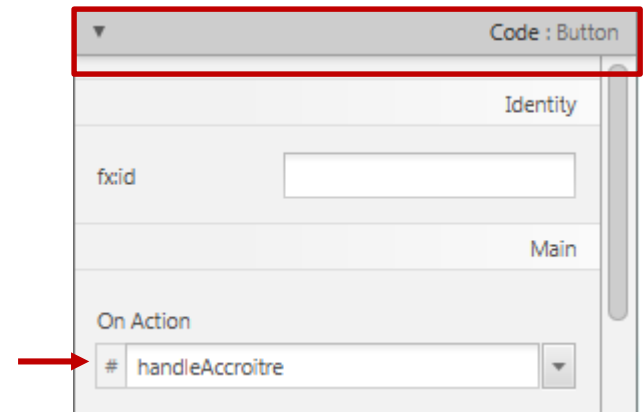
```
private void handleDiminuer(){  
}
```



# Contrôler les événements en JAVA

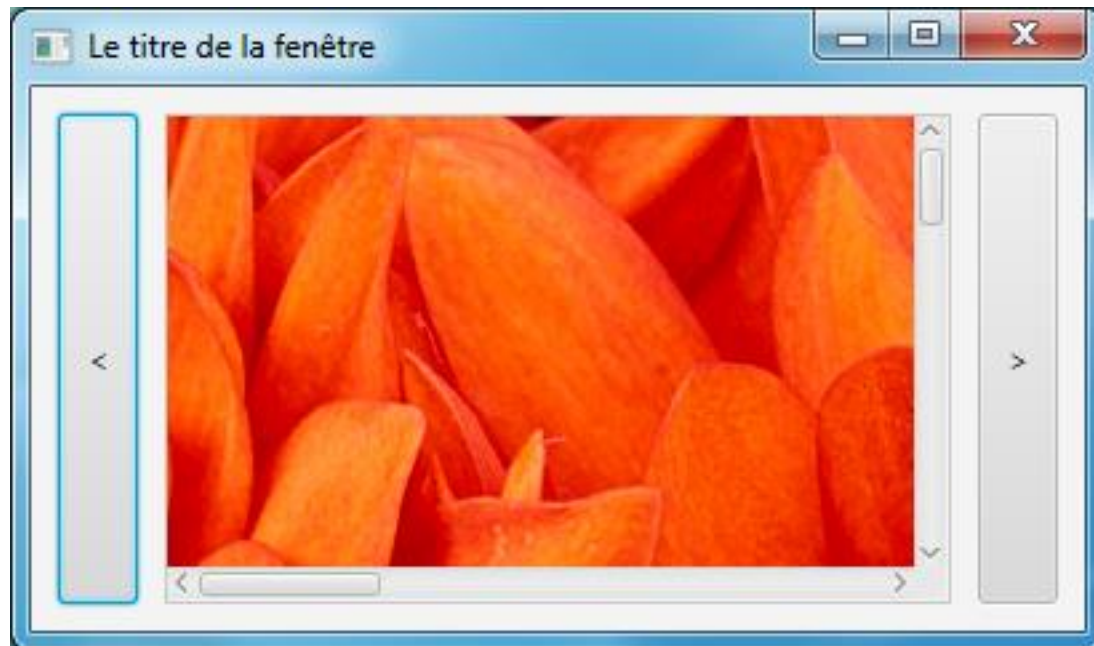
Sur chaque bouton, on va définir, via l'inspecteur de propriété dans la rubrique « Code » quelle méthode (exportée par @FXML du contrôleur) doit être associée à quel événement.

Pour l'exemple, on choisit d'associer, à une action quelconque sur le bouton « Accroitre », la méthode « handleAccroitre ».



# TD

- Créez une interface FXML ayant une première zone à gauche de 30 pixels de large contenant un bouton « < » sur toute sa hauteur, puis au milieu une image scrollable et enfin une dernière zone à droite de 30 pixel de large contenant un bouton « > » sur toute sa hauteur.



# TD

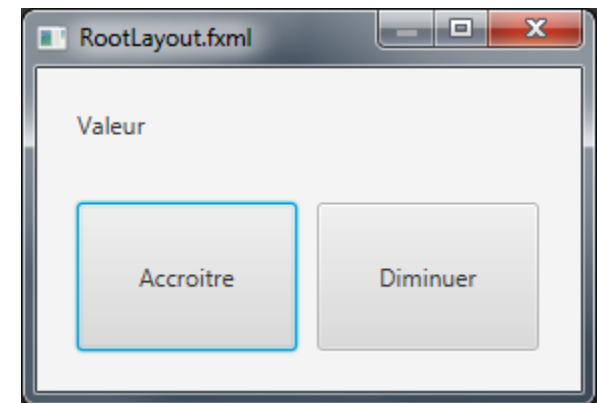
- Créez une interface FXML ayant une première zone à gauche de 30 pixels de large contenant un bouton « < » sur toute sa hauteur, puis au milieu une image scrollable et enfin une dernière zone à droite de 30 pixel de large contenant un bouton « > » sur toute sa hauteur.
- Créez un contrôleur affichant dans la console (via `system.out.println`) « even right » quand on clique sur le bouton droit et « even left » quand on clique sur le bouton gauche.
- Reliez les deux puis testez.

# Accéder aux valeurs des champs

Dans notre exemple, pour que les boutons puissent modifier la valeur affichée dans le label, il faut rendre accessible le label depuis chaque méthode de la classe « controller ».

On va donc déclarer le « Label » comme variable globale de la classe. Remarque : En programmation objet, on dira que c'est un attribut de la classe. Cela donne en Java :

```
public class Controller {  
    @FXML  
    private Label compteurLabel;  
    ...  
}
```

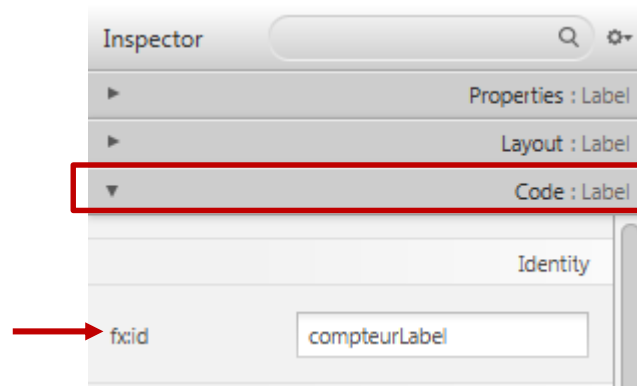




# Accéder aux valeurs des champs

La déclaration du « Label » comme attribut, le rend accessible des méthodes mais ne dit pas encore qu'il doit être mis en lien avec celui prévu par le fichier FXML. On utilise donc SceneBuilder pour définir le lien.

Via l'inspecteur de « Code » sur le label, on définit dans la rubrique « Identity » la propriété « fx:id » en indiquant le nom de la variable définit dans le « controller ».



# Méthodes de l'objet « Label »

Pour agir en java sur l'objet « Label », il faut utiliser ses méthodes. La javadoc (accessible via une recherche Google sur les mots clef « javadoc javafx label ») nous donne la liste des méthodes définies dans la classe « Label » ou dans son arbre d'héritage :

javafx.scene.control

## **Class Label**

java.lang.Object  
  javafx.scene.Node  
    javafx.scene.Parent  
      javafx.scene.layout.Region  
        javafx.scene.control.Control  
          javafx.scene.control.Labeled  
            javafx.scene.control.Label

javafx.scene.control

## **Class Button**

java.lang.Object  
  javafx.scene.Node  
    javafx.scene.Parent  
      javafx.scene.layout.Region  
        javafx.scene.control.Control  
          javafx.scene.control.Labeled  
            javafx.scene.control.ButtonBase  
              javafx.scene.control.Button

# Méthodes de l'objet « Label »

L'objet « Label », dispose, entre autre, des méthodes :

String      getText() Gets the value of the property text.

void      setText(String value) Sets the value of the property text.

## setText

```
public final void setText(String value)
```

Sets the value of the property text.

### Property description:

The text to display in the label. The text may be null.

## getText

```
public final String getText()
```

Gets the value of the property text.

### Property description:

The text to display in the label. The text may be null.

# Contrôler les événements en JAVA

```
import javafx.fxml.FXML;
import javafx.scene.control.Label;
```

```
public class Controller {
```

```
    @FXML
```

```
    private Label compteurLabel;
```

```
    private int compteurVal=0;
```

```
    public Controller() {
```

```
        System.out.println("call Controller : le  
constructeur");
```

```
    }
```

```
    @FXML
```

```
    private void initialize() {
```

```
        System.out.println("call Controller : initialize");
        compteurLabel.setText(Integer.toString(compteurVal))
        ;
```

```
    }
```

```
    @FXML
```

```
    private void handleAccroitre(){
```

```
        System.out.println("call Controller :  
handleAccroitre");
```

```
        compteurVal++;
```

```
        compteurLabel.setText(Integer.toString(compteurVal));
```

```
    }
```

```
    @FXML
```

```
    private void handleDiminuer(){
```

```
        System.out.println("call Controller :  
handleDiminuer");
```

```
        compteurVal--;
```

```
        compteurLabel.setText(Integer.toString(compteurVal));
```

```
    }
```

# TD

- Modifiez le contrôleur du TD précédent afin qu'il affiche dans la console un compteur d'image allant de 0 à 4, l'action left décrémentant le compteur (avec retour à 4 quand on devient négatif), l'action right incrémentant le compteur (avec retour à 0 quand on dépasse 4)
- Créer dans le contrôleur un objet « ImageView » que l'on rattachera à celui du fichier FXML.

# TD

- Créez comme attribut privé du contrôleur, un tableau de 5 objets « Image ».
- Créez une variable de type String nommée « dir » contenant l'URL d'un répertoire local contenant 5 images JPG. **Attention** : sous Windows pour accéder au répertoire « C:\Users\Public\Pictures\Sample Pictures\ » il faut utiliser l'URL : « file:/c:/Users/Public/Pictures/Sample%20Pictures/ »
- Dans l'initialisation du contrôleur, remplir le tableau des 5 images en créant des objets images construits à partir de l'URL de chaque fichier JPG. (On concaténera « dir » avec le nom du fichier pour obtenir l'URL complète.)

**Remarque** : La classe « ImageView » dispose de la méthode « setImage(...) » permettant de changer l'image visualisée par l'image (Objet de type Image) passée en paramètre.

- Modifier le code des Handler d'événement afin de changer l'image affichée au lieu d'afficher le numéro dans la console.

# Plan du cours

- Introduction
- Programmation Java
- Le FXML
- Utilisation d'un RAD
- Disposition adaptative
- Traitement des événements
- **Application multifenêtres**
- Annexes



# Application multi-fenêtres

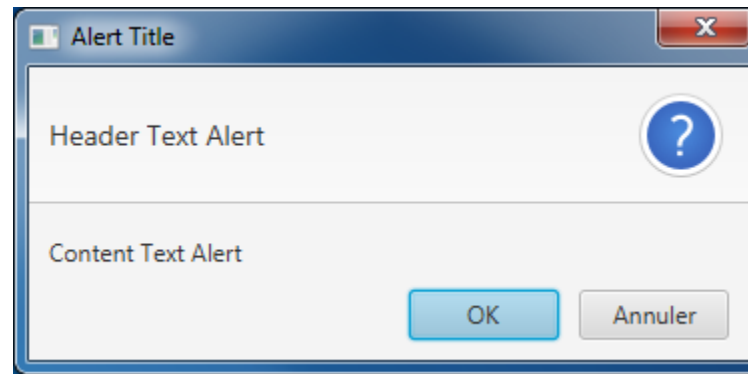
Par application multi-fenêtres, cela sous-entend que depuis la fenêtre principale, certains événements doivent déclencher l'ouverture d'autres fenêtres filles ou des boîtes de dialogues.

On va donc dans le « Controller » écrire du code java pour créer une nouvelle fenêtre.

# Boite de dialogue

## Fenêtre d'alerte

JavaFx propose un objet « Alert » permettant de créer une fenêtre de dialogue au look préétabli.

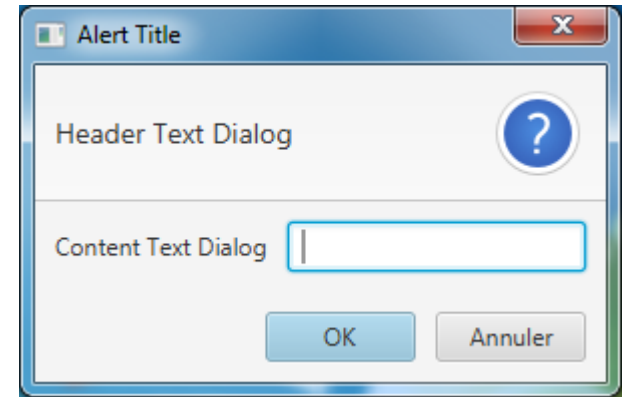


```
Alert alert = new Alert(AlertType.INFORMATION);  
alert.setTitle("Alert Title");  
alert.setHeaderText("Header Text Alert");  
alert.setContentText("Content Text Alert");  
alert.showAndWait();
```

# Boite de dialogue

## Fenêtre de saisie

JavaFx propose un objet « `TextInputDialog` » permettant de créer une fenêtre de dialogue au look préétabli afin de saisir une valeur. Il s'utilise comme l'objet « `Alert` ».



```
TextInputDialog dial = new TextInputDialog();  
dial.setTitle("Alert Title");  
dial.setHeaderText("Header Text Dialog");  
dial.setContentText("Content Text Dialog");  
Optional<String> value=dial.showAndWait();
```

# Boite de dialogue

## Fenêtre de saisie

L'objet de type « `Optional<String>` » dispose de la méthode boolean « `isPresent()` » permettant de savoir si on a validé la saisie.

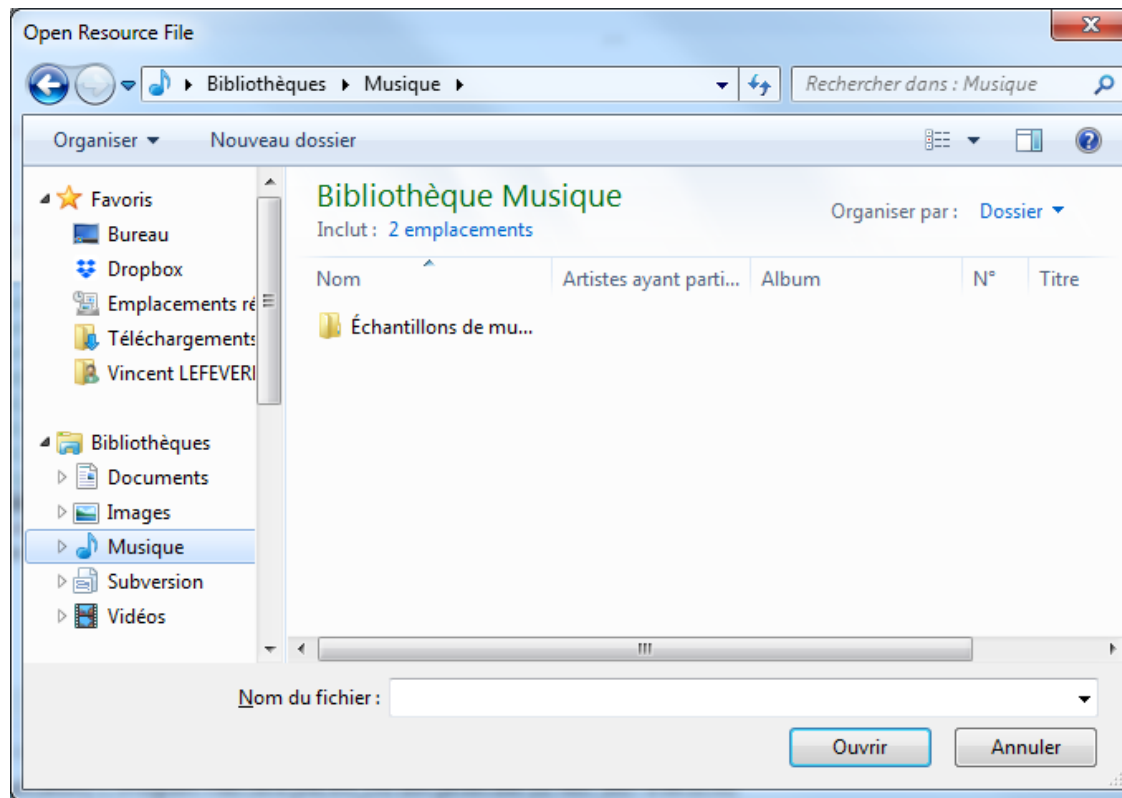
```
if (value.isPresent()) {
```

Il dispose également de la méthode « `get()` » pour récupérer la valeur saisie (de type « `String` »).

```
String chaine=value.get();
```

# Fenêtre sélection de fichier : FileChooser

Parmi les fenêtres de dialogue au look préétabli, JavaFX propose l'accès à la fenêtre de sélection de fichier de l'interface graphique de votre système d'exploitation via l'objet FileChooser.



# Fenêtre sélection de fichier : FileChooser

```
FileChooser fileChooser = new FileChooser();  
fileChooser.setTitle("Open Resource File");  
File file =fileChooser.showOpenDialog(null);
```

On commence par créer l'objet « FileChooser », puis on fixe le titre de la fenêtre de dialogue via la méthode « setTitle », enfin on fait apparaître la fenêtre de sélection via la méthode « showOpenDialog ».

# Fenêtre sélection de fichiers : FileChooser

L'objet « FileChooser » dispose également d'un accès à la fenêtre de sélection multiple via la méthode « showOpenMultipleDialog ».

```
FileChooser fileChooser = new FileChooser();  
fileChooser.setTitle("Open Resource Files");  
[ List<File> list =  
    fileChooser.showOpenMultipleDialog(null);
```

On récupère alors la liste des fichiers sélectionnés qu'il ne reste plus qu'à parcourir comme une liste pour récupérer les informations de chaque fichier sélectionné.

# Fenêtre sélection de répertoire : DirectoryChooser

L'objet « DirectoryChooser » permet quant à lui de sélectionner un répertoire.

```
DirectoryChooser dirChooser =  
    new DirectoryChooser();  
dirChooser.setTitle("Open Resource Dir");  
File dir=dirChooser.showDialog(null);
```

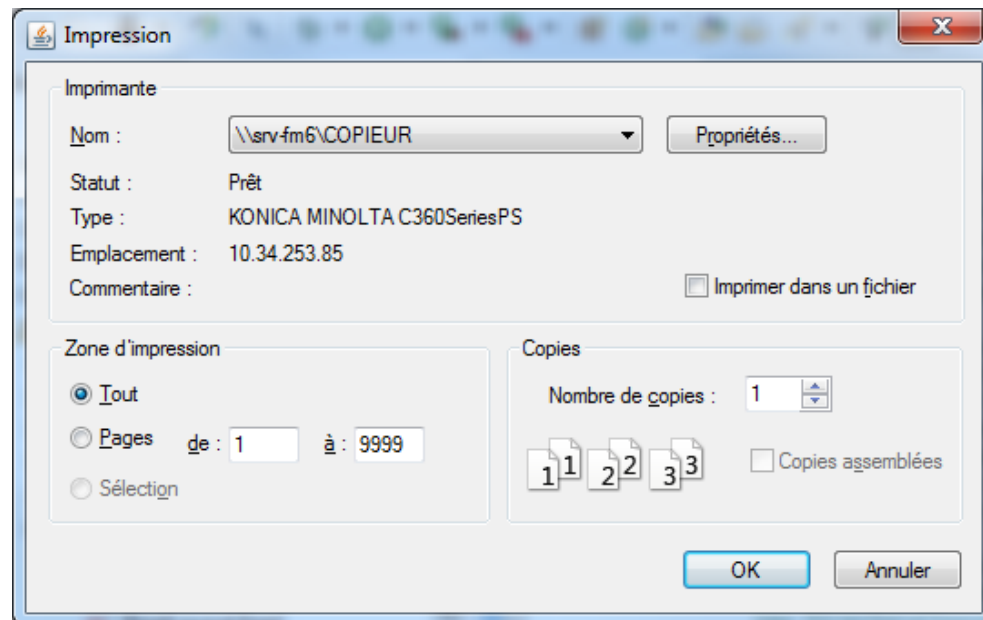
On commence par créer l'objet « DirectoryChooser », puis on fixe le titre de la fenêtre de dialogue via la méthode « setTitle », enfin on fait apparaître la fenêtre de sélection via la méthode « showDialog ».



# Fenêtre sélection d'impression : showPrintDialog

L'objet « PrinterJob » permet de gérer les impressions et offre, via sa méthode « showPrintDialog », l'accès à la fenêtre de sélection d'impression du système d'exploitation.

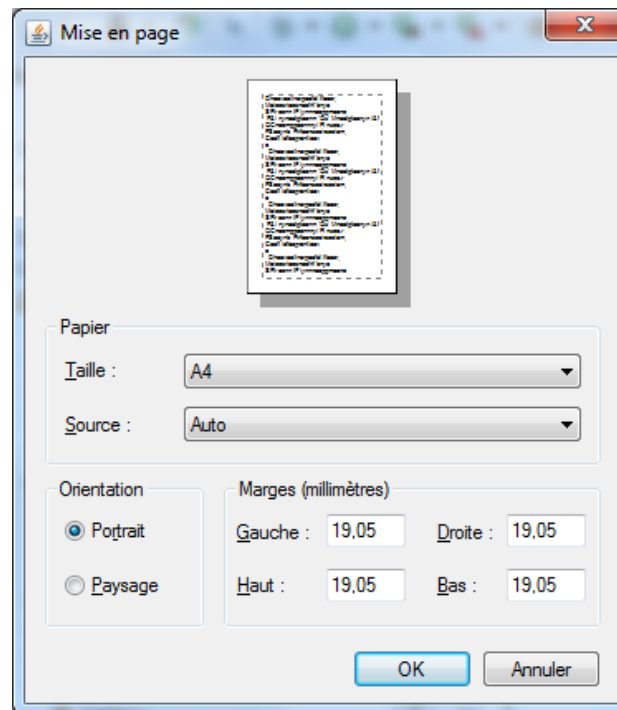
```
PrinterJob job = PrinterJob.createPrinterJob();  
job.showPrintDialog(null);
```



# Fenêtre sélection d'impression : showPageSetupDialog

L'objet « PrinterJob » offre également, via sa méthode « showPageSetupDialog », l'accès à la fenêtre de mise en page.

```
PrinterJob job = PrinterJob.createPrinterJob();  
job.showPageSetupDialog(null);
```



# TD

- Créer une application avec une bar de menus contenant l'item « Open » dans le menu « File ».
- Quand on clique sur l'item « Open », la fenêtre de sélection d'un fichier doit s'ouvrir.
- Ajouter l'item « Save » dans le menu « File ».
- Regardez, dans la javadoc ou dans les méthodes qu'éclipse propose pour la classe FileChooser, comment avoir une fenêtre de sélection de fichier avec un bouton « Enregistrer » au lieu d'un bouton « Ouvrir ».

# Fenêtre construite

Il est également possible de créer une fenêtre définie par un fichier fxm1. La méthode de traitement de l'événement devra avec les mots clef : « **throws Exception** » comme on l'avait fait pour la méthode « start » dans la classe « MainApp ».

Remarque : la Méthode devra créer l'estrade (« Stage ») puisqu'elle ne la reçoit pas en paramètre à la différence de « start ».

Enfin, on affichera la fenêtre et procédant à l'attente des événements liés à cette fenêtre en appelant la méthode « showAndWait() ».

# Création de l'estrade

```
Stage dialogStage = new Stage();
```

...

```
dialogStage.showAndWait();
```

# Fenêtre construite

On obtient donc le code JavaFX suivant :

**@FXML**

**private void handle...() throws Exception {**

Stage dialogStage = **new Stage()** ;

FXMLLoader loader = **new FXMLLoader()** ;

loader.setLocation(MainApp.class.getResource("DialogLayout.fxml")) ;

AnchorPane Layout = (AnchorPane) loader.load() ;

Scene scene = **new Scene(Layout)** ;

dialogStage.setScene(scene) ;

dialogStage.setTitle("Dialog Window") ;

dialogStage.showAndWait() ;

**}**

# Fenêtre construite

Comme la seconde fenêtre doit également gérer des événements, il faudra créer une classe servant de « controller » au fichier FXML comme on a déjà procédé pour la fenêtre principale.

# TD

- Créer une application ayant sur la première ligne un bouton « Afficher Code HTML » et sur la ligne du dessous un objet d'édition HTML (en utilisant l'Objet JavaFX « HTMLEditor »).
- Quand on presse le bouton, une seconde fenêtre affichant uniquement un objet JavaFX « TextArea » apparaîtra.



# Fenêtre modale

**Remarque** : Quand la fenêtre secondaire est ouverte, on peut encore cliquer dans la fenêtre principale (et donc créer plusieurs fenêtres secondaires). Si cela ne convient pas, il faut ajouter du code java pour rendre la fenêtre secondaire « modale » (c'est-à-dire bloquer les événements vers la fenêtre principale).

Le même problème survient dans le code de création des fenêtres de sélection de fichiers, de répertoire, d'imprimante, ... que l'on vous a précédemment présenté. Cela vient de l'utilisation du paramètre « **null** » qui rend la fenêtre non modale. Il faudrait utiliser le « **primaryStage** » à la place.

# Fenêtre modale

Pour construire une fenêtre modale, il faut ajouter au niveau de la création de la fenêtre deux appels de méthode JavaFX :

```
dialogStage.initModality(Modality.WINDOW_MODAL);  
dialogStage.initOwner(primaryStage);
```

Le **problème** vient que « primaryStage » est une variable de la classe MainApp et non une variable de la classe Controller. Il va donc falloir en plus programmer un passage de valeur entre l'objet créé dans « launch » à partir de la classe « MainApp » et l'objet créé par le FXMLoader à partir de la classe « Controller ».

# Fenêtre modale

On crée donc dans la classe « controller » un attribut qui doit être accessible hors de l'objet. Cela se fait en java par :

①

```
public class Controller {  
    public Stage primaryStage;  
    ...  
}
```

Ainsi, la création de la fenêtre modale fonctionnera du moment que l'on initialise « primaryStage » dans la méthode « start » de « MainApp »

# Fenêtre modale

Dans la méthode « start », après que le « loader » ait chargé le fichier FXML, on peut récupérer l'objet de type « Controller » créé par la méthode « load ». En javafx, cela se fait par l'appel de la méthode « getController » du « loader » :

```
Controller controller=loader.getController();
```

Ensuite, on affecte la variable « primaryStage » à l'attribut « primaryStage » de controller.

```
controller.primaryStage=primaryStage;
```

# Fenêtre modale

```
public class MainApp extends Application {
```

```
    public void start(Stage primaryStage) throws Exception {
```

```
        primaryStage.setTitle("Le titre de la fenêtre");
```

```
        FXMLLoader loader = new FXMLLoader();
```

```
        loader.setLocation(MainApp.class.getResource("RootLayout.fxml"));
```

```
        AnchorPane rootLayout = (AnchorPane) loader.load();
```

```
        Controller controller=loader.getController();
```

② → controller.primaryStage=primaryStage;

```
        Scene scene = new Scene(rootLayout);
```

```
        primaryStage.setScene(scene);
```

```
        primaryStage.show();
```

```
}
```

# Fermeture de fenêtre

Pour fermer une fenêtre dans le traitement d'un événement, il faut également pouvoir, dans le contrôleur, accéder à l'objet de type « Stage » et appeler sa méthode « close() ».

# Fermeture de fenêtre

Donc pour pouvoir fermer la seconde fenêtre depuis le « **DialogController** », il faut ajouter:

```
DialogController controller=loader.getController();  
controller.dialogStage=dialogStage;
```

dans le code de création de la fenêtre à partir de son fichier FXML.

# Fermeture de fenêtre

Et dans la classe « **DialogController** », on ajoute l'attribut public « **dialogStage** » de type « **Stage** » :

```
public class DialogController {  
    public Stage dialogStage;  
    ...  
}
```

Enfin dans le handle de l'événement, on utilise l'instruction :

```
dialogStage.close();
```



# TD

- Modifiez la fenêtre de chargement de l'application ayant un menu avec l'item « Ouvrir » afin qu'elle soit une fenêtre modale.
- Ajouter au menu « Fichier » un item « Quitter » qui ferme l'application en fermant la fenêtre principale.

# TD

- Modifiez l'application d'éditeur HTML afin d'afficher dans le « TextArea » de la second fenêtre, le code HTML créé dans la fenêtre principale.

## Remarque :

La méthode « **setText(...)** » de l'Objet JavaFX « TextArea » permet de fixer le texte devant y apparaître.

La méthode « **getHtmlText()** » de l'Objet JavaFX « HTMLEditor » renvoie le code HTML correspondant à son contenu.

# Plan du cours

- Introduction
- Programmation Java
- Le FXML
- Utilisation d'un RAD
- Disposition adaptative
- Traitement des événements
- Application multifenêtres
- **Annexes**

# Annexe : les champs listes

JavaFX propose plusieurs champs de saisies utilisant des listes pour définir l'ensemble de leurs valeurs :

ListView	ChoiceBox
TableColumn	ComboBox

On peut placer ces champs dans l'interface via « Scene Builder », mais il faudra de la programmation java pour agir sur la liste des valeurs sélectionnables de ces champs.

On commence en procédant comme on l'a vu pour un champs « Label » ou « Button » : on le déclare dans la classe « Controller » associée à la description FXML.

# Annexe : les champs listes

La liste que l'on a créé sous le nom de variable « data » peut ensuite être manipuler comme une « ArrayList » ou un « LinkedList » via les méthodes :

- `add(valeur)`
- `clear()`
- `get(index)`
- `isEmpty()`
- `remove(index)`
- `set(index, valeur)`
- `size()`

Revoir éventuellement  
le chapitre  
sur les collections.

# Annexe : les champs listes

Il faut ensuite indiquer à l'objet graphique la liste de ses items via la méthode « `setItems(laList)` ». Ce qui donne dans notre exemple :

```
ObservableList<String> data = FXCollections.observableArrayList(
    "Lundi", "Mardi", "Mercredi");
data.add("Jeudi");
data.add("Vendredi");
monChamps.setItems(data);
monChamps.getSelectionModel().selectionModelProperty()
    .set(SelectionMode.SINGLE);
```

*La dernière instruction, qui n'existe que pour la ListView, permet de fixer le mode de sélection des valeurs. On a le choix entre « **SINGLE** » ou « **MULTIPLE** ».*

# Annexe : les champs listes

Il faut maintenant créer un gestionnaire spécifique pour traiter les changements de sélection.

On crée donc une nouvelle classe que l'on nommera par exemple « ListViewController » pour gérer une « ListView ».

```
public class ListViewController implements  
ChangeListener<String> {
```

```
    public void changed(ObservableValue<? extends String> observable,  
        String oldValue, String newValue) {  
    }
```

```
}
```

# Annexe : les champs listes

On peut ajouter du code dans la méthode « changed » et récupérer via les paramètres l'ancienne valeur sélectionnée (oldValue) ou la nouvelle valeur sélectionnée (newValue).

Il faut ensuite indiquer à notre « ListView » quel objet gère les changement de sélection. Ainsi à chaque changement de sélection dans notre « ListView », il appellera la méthode « changed ».



# Annexe : les champs listes

On retourne dans la méthode « initialize() » de la classe « Controller » et on ajoute le code suivant :

```
ListViewController monListViewController=  
    new ListViewController();  
maListView.getSelectionModel().  
    selectedItemProperty().  
        addListener(monListViewController);
```

La première ligne crée l'objet correspondant à notre nouvelle class « ListViewController ». La ligne suivante définit pour notre champs « ListView » quel objet en gère les événements de sélection.

# Annexe : les champs listes

Il est non seulement possible de savoir ce que l'on a sélectionner dans la liste mais également d'agir sur la sélection.

Le champs nous offre les méthodes suivantes :

- `.getSelectionModel().selectFirst()`
- `.getSelectionModel().selectPrevious()`
- `.getSelectionModel().selectNext()`
- `.getSelectionModel().selectLast()`
- `.getSelectionModel().select(index)`

**Remarque** : la sélection d'un nouvelle élément, via ces méthodes, déclenche, comme une sélection graphique, l'appel de « changed » dans notre « ListViewController ».

# Annexe : Exemple de TD

- Créer une application ayant un « ListView » de tous les mois de l'année et affichant dans la console le changement de sélection d'item (sélection simple).
- Ajouter deux boutons, l'un (« - ») pour remonter dans la liste et l'autre (« + ») pour descendre dans la liste.

# Annexe : les champs arbres

JavaFX propose plusieurs champs de saisies utilisant un arbre de valeurs :

TreeView

TreeTableColumn

Comme dans le cas des champs listes, on peut les placer dans l'interface via « Scene Builder », mais il faudra de la programmation java pour agir sur le contenu de l'arbre.

On commence comme précédemment, en les déclarant dans la classe « Controller » associée à la description FXML.

# Annexe : les champs arbres

Pour définir les valeurs, il suffit d'ajouter du code dans la méthode « initialize » de la classe « Controller ».

En premier lieu, on crée un nœud JavaFX pour notre arbre :

```
Treeltem<String> rootItem = Treeltem("racine");
```

La chaîne de caractères passée au constructeur du nœud précise le nom que l'on veut associer à ce nœud.

Ensuite, on peut, éventuellement, utiliser la méthode « **getChildren()** » afin d'accéder à la liste de ces nœuds fils et pouvoir ainsi y ajouter des fils via la méthode « **add(node)** ». La variable *node* doit également être créée de type « **Treeltem**<String> ».

# Annexe : les champs arbres

Il faut ensuite indiquer à l'objet graphique la liste de ses items via la méthode « `setItems(laList)` ». Ce qui donne dans notre exemple :

```
Treeltem<String> rootItem = new Treeltem<String>("Racine");  
Treeltem<String> node = new Treeltem<String>("Noeud 1");  
rootItem.getChildren().add(node);  
monTreeView.setRoot(rootItem);
```

# Annexe : les champs arbres

Il faut maintenant créer un gestionnaire spécifique pour traiter les changements de sélection dans notre arbre.

On crée donc une nouvelle classe que l'on nommera par exemple « `TreeViewController` » pour gérer une « `TreeView` ».

```
public class TreeViewController implements  
ChangeListener<String> {  
  
    public void changed(  
        ObservableValue<? extends TreeItem<String>> observable,  
        TreeItem<String> oldValue,  
        TreeItem<String> newValue) {  
  
    }  
  
}
```

# Annexe : les champs arbres

Comme dans le cas des listes, on peut ajouter du code dans la méthode « `changed` » et récupérer via les paramètres l'ancienne valeur sélectionnée (`oldValue`) ou la nouvelle valeur sélectionnée (`newValue`).

**Remarque** : le type « `String` » (utilisé dans le cas d'une liste de chaînes de caractères) devient le type « `Treeltem<String>` » (correspondant à un nœud de l'arbre de chaîne de caractères).

**Remarque** : la méthode « `getValue()` » d'un objet de type « `Treeltem<String>` » permet de récupérer la « `String` » associée au nœud.



# Annexe : les champs arbres

Il faut ensuite indiquer à notre « `TreeView` » quel objet gère les changements de sélection. Ainsi à chaque changement de sélection dans notre « `TreeView` », il appellera la méthode « `changed` ».

On retourne dans la méthode « `initialize()` » de la classe « `Controller` » et on ajoute le code suivant :

```
TreeViewController monTreeViewController=  
    new TreeViewController();  
monTreeView.getSelectionModel().  
    selectedItemProperty().  
        addListener(monTreeViewController);
```

# Annexe : Exemple de TD

- Créer une application ayant un « TreeView » possédant seulement une racine « racine » et affichant dans la console et mémorisant dans une variable le nœud sélectionné dans l'arbre.
- Dans « Controller » ajouter deux méthodes de traitement d'événement : l'une pour demander une valeur (via une fenêtre popup) et ajouter, au nœud sélectionné, un nœud fils de cette valeur, l'autre pour supprimer tous les nœuds fils du nœud sélectionné.
- Via « Scene Builder » ajouter un menu contextuel dans le « TreeView » qui propose deux items : « add » et « clear » associés aux deux méthodes précédentes.

# Annexe : Événement périodique

Dans certaine application, on a besoin d'avoir un événement qui se déclenche périodiquement pour, par exemple, gérer une animation.

On doit alors utiliser le code suivant :

```
Timeline timeline;  
timeline = new Timeline();  
KeyFrame kfl=new KeyFrame(Duration.millis(1000), ctrl );  
timeline.getKeyFrames().add(kfl);  
timeline.setCycleCount(Timeline.INDEFINITE);  
timeline.play();
```

# Annexe : Événement périodique

La méthode « `setCycleCount` » permet de déterminer le nombre de fois que l'événement doit être appelé, la constante « `Timeline.INDEFINITE` » servant la répétition continue.

Dans l'exemple, on provoque l'événement toutes les secondes, en créant un « `KeyFrame` » initialisé à 1000ms.

La méthode « `play` » lance le timer. Et le timer peut ensuite être stoppé via la méthode « `stop` » si nécessaire.

# Annexe : Événement périodique

La variable « ctrl » doit alors contenir l'instance du contrôleur qui va gérer l'événement via sa méthode :

**handle(ActionEvent event)**

Le contrôleur doit implémenter :

**EventHandler<ActionEvent>**

# Annexe : Événement périodique

Le code de la classe TimerController est donc :

```
public class TimerController implements EventHandler<ActionEvent> {  
  
    public void handle(ActionEvent event) {  
    }  
}
```

Et avant de créer le timer, on déclare ctrl via :

```
TimeController ctrl = new TimeController();
```

# Annexe : Exemple de TD

- Créez un fichier FXML pour une application ayant un « Label » et deux « Button ». Le premier bouton sera appelé « start » et le second « stop ».
- Créez le fichier MainApp.java qui charge le fichier FXML.
- Dans la méthode « initialize » du « controller », faites l'initialisation du timer (type Timeline) pour un appel de méthode toutes les 100ms.
- Faîtes gérer le démarrage et l'arrêt du chronomètre via les boutons « start » et « stop ».
- Le « TimeController » affiche le temps (en 1/10 de seconde) via le label auquel on lui aura permis l'accès.

# Annexe : ajout d'un Événement en Java

Il est parfois intéressant de construire une partie de l'interface graphique via la programmation en Java. On doit alors également associer un événement à un objet par programmation en Java.

Soit un objet graphique JavaFX stocké dans une variable « obj », on utilise alors la méthode « `addEventHandler(event, ctrl)` ».

A chaque fois que l'événement indiqué par « event » se produira, la méthode « `public void handle(Event event)` » du contrôleur « ctrl » sera appelée.



# Annexe : ajout d'un Événement en Java

Les événements peuvent prendre, entre autre, les valeurs suivantes :

Event.ANY

MouseEvent.MOUSE\_CLICKED

MouseEvent.MOUSE\_ENTERED

MouseEvent.MOUSE\_EXITED

MouseEvent.ANY

KeyEvent.KEY\_PRESSED

KeyEvent.ANY

...

Voir la javadoc pour plus de détails

# Annexe : ajout d'un Événement en Java

Le contrôleur que l'on associe à l'événement (dans variable « ctrl ») doit implémenter :

**EventHandler<Event>**

et doit contenir la méthode :

**handle(Event event)**

qui sera appelé pour chaque événement lui correspondant.

# Annexe : Exemple de TD

- Créez une application dont l'interface est décrite en FXML ayant un menu ainsi qu'un « Pane » dans lequel on placera des objet JavaFX par programmation Java.
- Dans la méthode « Initialize » de « Controller », créez, dans le « Pane », un tableau de 4 sur 4 cercles construits via l'objet « Circle ».

*Remarque : On utilisera les méthodes setRadius, setCenterX, setCenterY et setFill pour fixer les paramètres du cercle : rayon, centre et couleur de remplissage*

# Annexe : Exemple de TD

- Créez la classe « CircleController » pour gérer les événements d'un cercle. Ajoutez à cette classe deux attributs publics de type « int » pour mémoriser les indices du cercle dont il gère les événements.

*Remarque : sa méthode « handle » affichera la valeur des deux indices.*

- Ajoutez pour chaque cercle, une instance de « CircleController » (ayant ses valeurs d'indice) comme gestionnaire d'événements.