

Programmation Orientée Objet Java

M.WANE

Ingénieur Informaticien

Analyste, Concepteur

Développeur d'Applications

Cours 1 : Interfaces Graphiques avec Java FX

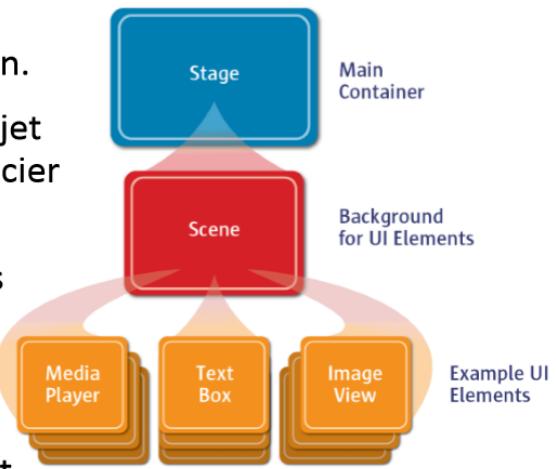
Java FX

- Java FX est la dernière génération de bibliothèques qui permet de créer des interfaces graphiques de qualité pour les applications java DeskTop, Mobile et Web.
- Avec l'apparition de Java 8 en mars 2014, JavaFX devient la bibliothèque de création d'interface graphique officielle du langage Java
- Le développement de son prédecesseur SWING étant abandonné (sauf pour les corrections de bogues).

Structure d'une application JavaFX

- Une application JavaFX se compose de d'une hiérarchie de composants :
 - Un composant **Stage** qui représente la fenêtre principale de l'application. A un instant donné, le composant Stage affiche une scène.
 - Le composant **Scene** qui permet d'afficher tout ce qui devrait apparaître dans l'application. L'objet Scene contient des composants graphiques organisés d'une manière hiérarchique.
 - Les composants graphiques sont des objets qui peuvent être de différents types :
 - Des éléments de contrôles utilisateur: **Label, TextField, ListView ...**
 - Des formes graphiques : **Circle, Rectangle, Line, ...**
 - Des médias: **ImageView, MediaView, etc...**
 - Des graphiques : **PieChart, LineChart, BarChart, ...**
 - Les Layouts pour grouper les éléments pour assurer les mises en page : **BorderPane, Hbox, Vbox, GridPane, ...**

- Comment se présente une application *JavaFX* ?
(petite immersion avant de décrire plus en détail les concepts de base).
- L'application est codée en créant une sous-classe de **Application**.
- La fenêtre principale d'une application est représentée par un objet de type **Stage** qui est fourni par le système au lancement de l'application.
- L'interface est représentée par un objet de type **Scene** qu'il faut créer et associer à la fenêtre (**Stage**).
- La scène est composée des différents éléments de l'interface graphique (composants de l'interface graphique) qui sont des objets de type **Node**.
- La méthode **start()** construit le tout.



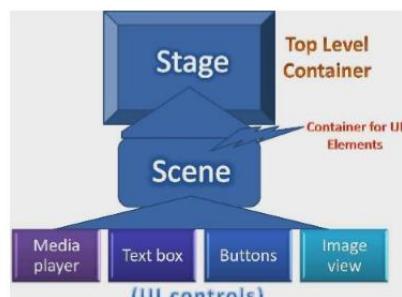
Métaphore de la salle de spectacle



- Les éléments structurels principaux d'une application *JavaFX* se basent sur la métaphore de la *salle de spectacle* (*theater*).

Remarque : En français, on utilise le terme 'scène' pour parler de l'endroit où se passe le spectacle (l'estrade, les planches) mais également pour parler de ce qui s'y déroule (jouer ou tourner une scène) ce qui peut conduire à un peu de confusion avec cette métaphore.

- **Stage** : L'endroit où a lieu l'action, où se déroule la scène
- **Scene** : Tableau ou séquence faisant intervenir les acteurs
- **Nodes** : Acteurs, figurants, éléments du décor, ... (éléments actifs/passifs)
- **Controls** : qui font partie de la scène en train d'être jouée.
- **Components**
- **Widgets**



Hello World [1]



- Une première application, le traditionnel *Hello World* !

```
public class HelloWorld extends Application {  
    //-----  
    @Override  
    public void start(Stage primaryStage) {  
        primaryStage.setTitle("My First JavaFX App");  
        BorderPane root = new BorderPane();  
        Button btnHello = new Button("Hello World");  
        root.setCenter(btnHello);  
        Scene scene = new Scene(root, 250, 100);  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
    //-----  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```



Hello World [2]



- JavaFX étant intégré à la plateforme de base *Java*, aucune librairie externe n'est nécessaire au fonctionnement de l'exemple précédent.
- Un certain nombre d'importations doivent cependant être effectuées (on y retrouve les classes principales *Application*, *Stage*, *Scene* ainsi que les composants *BorderPane* et *Button*) :
 - `import javafx.application.Application;`
 - `import javafx.scene.Scene;`
 - `import javafx.scene.control.Button;`
 - `import javafx.scene.layout.BorderPane;`
 - `import javafx.stage.Stage;`
- Dans cet exemple, la méthode `main()` lance uniquement la méthode statique `launch()` (qui est définie dans la classe *Application*).
 - Dans une application un peu plus complexe, elle pourrait effectuer d'autres opérations d'initialisation avant l'invoquer `launch()`.

En principe, Eclipse s'en charge automatiquement

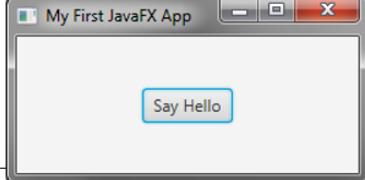


Traiter une action de l'utilisateur



- Dans l'exemple *Hello World*, pour que le clic sur le bouton déclenche une action, il faut traiter l'événement associé.
- La méthode `setOnAction()` du bouton permet d'enregistrer un *Event Handler* (c'est une interface fonctionnelle possédant la méthode `handle(event)` qui définit l'action à effectuer).

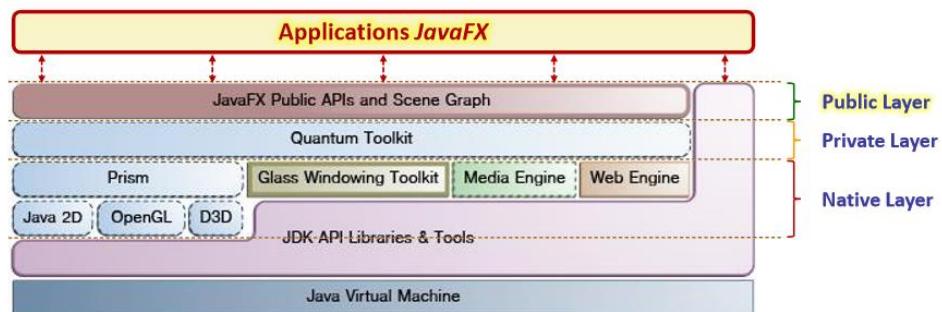
```
public void start(Stage primaryStage) {  
    primaryStage.setTitle("My First JavaFX App");  
    BorderPane root = new BorderPane();  
    Button btnHello = new Button("Say Hello");  
  
    btnHello.setOnAction(  
        event -> System.out.println("Hello World !"));  
  
    root.setCenter(btnHello);  
    Scene scene = new Scene(root, 250, 100);  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}
```



Architecture technique [1]



- L'architecture technique de la plateforme JavaFX est composée de plusieurs couches (*library stack*) qui reposent sur la machine virtuelle Java (JVM).

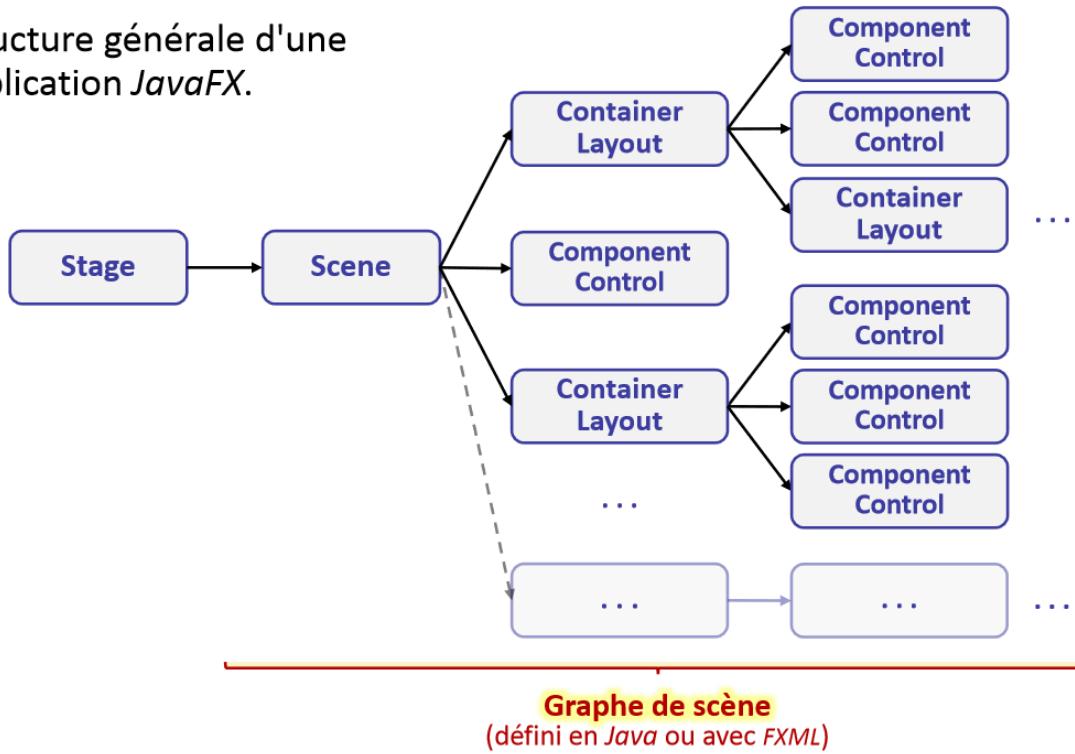


- Les développeurs ne devrait utiliser que la **couche (API) publique** car les couches inférieures sont susceptibles de subir des changements importants au fil des versions (sans aucune garantie de compatibilité).
- Les "briques" intermédiaires ne seront donc pas décrites dans ce cours.

Éléments d'une application



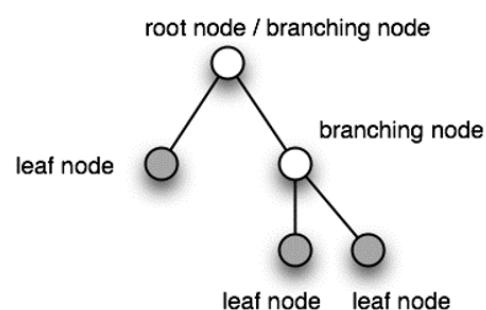
- Structure générale d'une application JavaFX.



Graphe de scène [1]



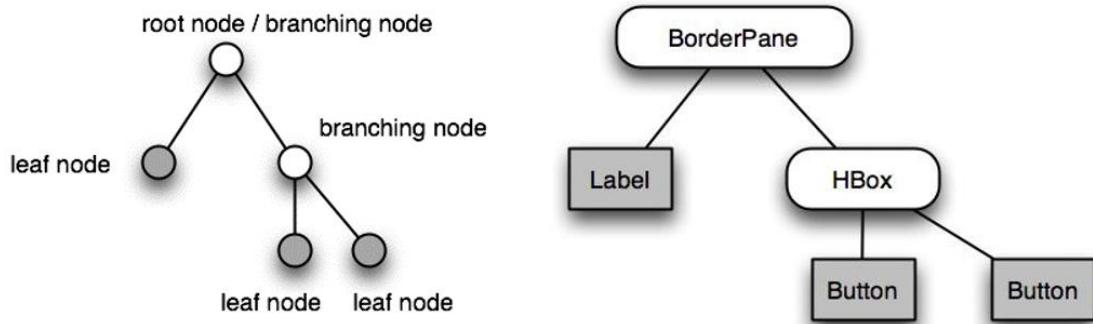
- Le **graphe de scène (scene graph)** est une notion importante qui représente la structure hiérarchique de l'interface graphique.
- Techniquement, c'est un graphe acyclique orienté (arbre orienté) avec :
 - une racine (*root*)
 - des nœuds (*nodes*)
 - des arcs qui représentent les relations parent-enfant
- Les nœuds (*nodes*) peuvent être de trois types :
 - Racine
 - Nœud intermédiaire
 - Feuille (*leaf*)



Graphe de scène [2]



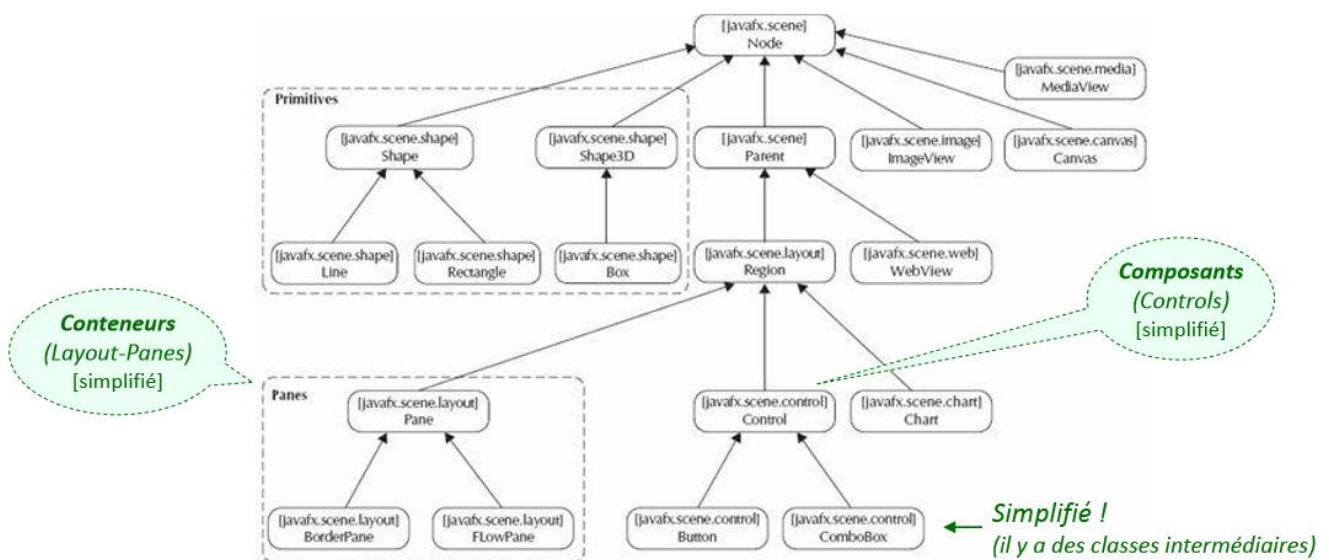
- Les **feuilles** de l'arbre sont généralement constitués de composants visibles (boutons, champs texte, ...) et les **nœuds intermédiaires** (y compris la **racine**) sont généralement des éléments de structuration (souvent invisibles), typiquement des conteneurs ou panneaux de différents types (*HBox*, *VBox*, *BorderPane*, ...).



Graphe de scène [3]



- Tous les éléments contenus dans un graphe de scène sont des objets qui ont pour classe parente la classe **Node**.
- La classe **Node** comporte de nombreuses sous-classes :



Graphe de scène [4]



- Parmi les sous-classes de **Node** on distingue différentes familles :

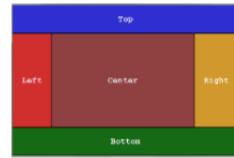
- Les **formes primitives (Shape)** 2D et 3D

⇒ `Line`, `Circle`, `Rectangle`, `Box`, `Cylinder`, ...



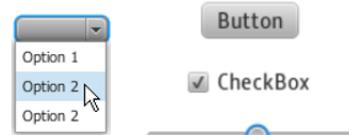
- Les **conteneurs (Layout-Pane)** qui se chargent de la disposition (*layout*) des composants enfants et qui ont comme classe parente **Pane**.

⇒ `AnchorPane`, `BorderPane`, `GridPane`, `HBox`, `VBox`, ...



- Les **composants standard (Controls)** qui étendent la classe **Control**.

⇒ `Label`, `Button`, `TextField`, `ComboBox`, ...



- Les **composants spécialisés** qui sont dédiés à un domaine particulier (par exemple : lecteur multimédia, navigateur web, etc.).

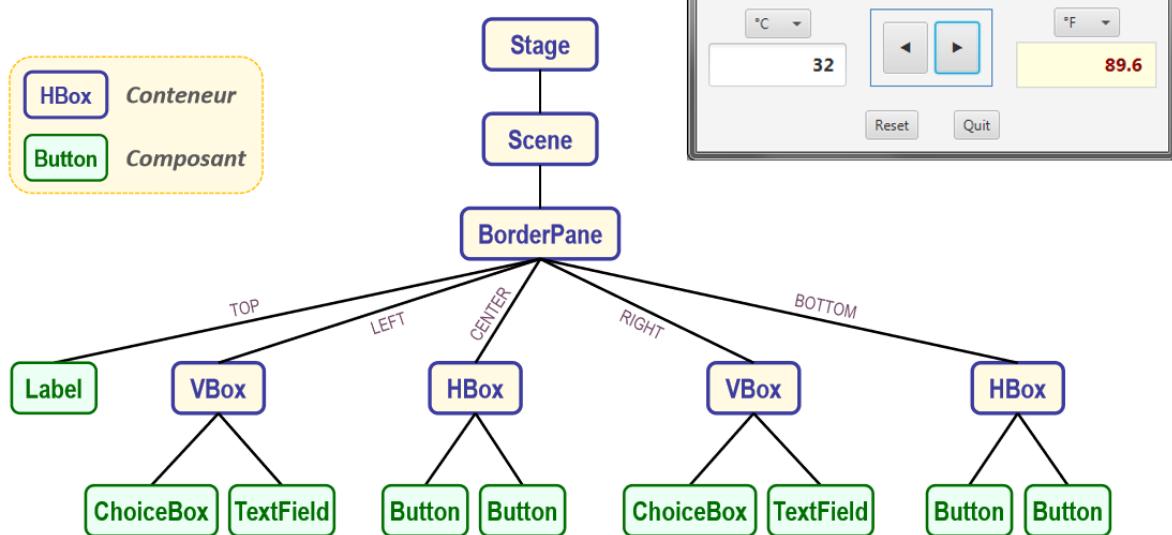
⇒ `MediaView`, `WebView`, `ImageView`, `Canvas`, `Chart`, ...



Graphe de scène [5]



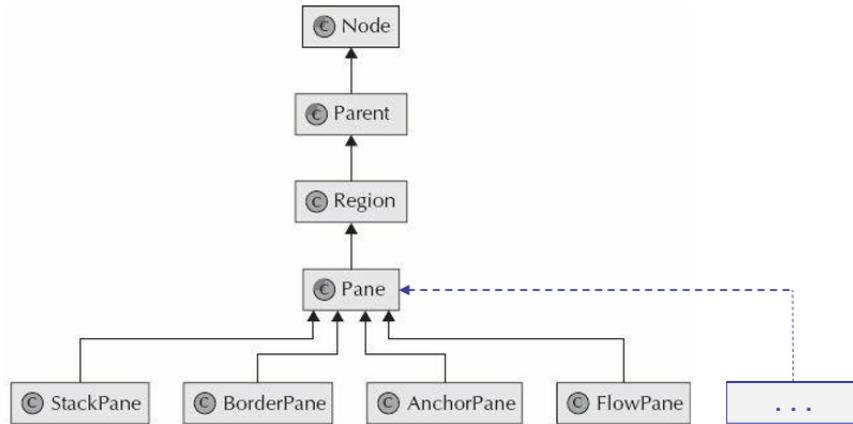
- Application *JavaFX* et son graphe de scène.



Conteneurs



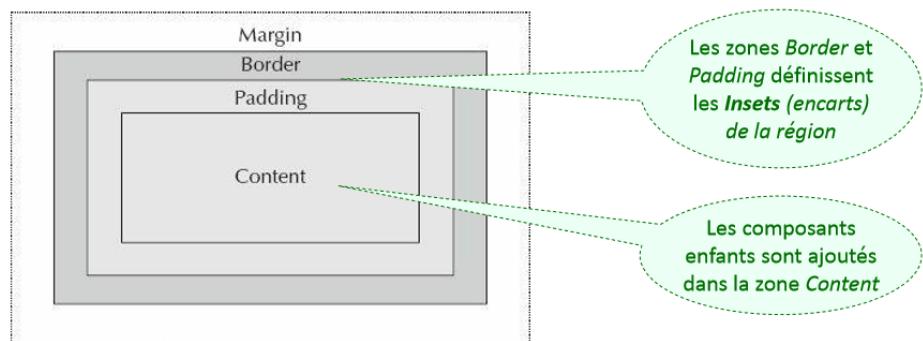
- Les **conteneurs (*Layout-Pane*)** représentent une famille importante parmi les sous-classes de **Node**. Ils ont pour classe parente **Pane** et **Region** qui possèdent de nombreuses propriétés et méthodes héritées par tous les conteneurs.



Region - Structure visuelle [1]



- La classe **Region** est la classe parente des composants (*Controls*) et des conteneurs (*Layout-Panes*).
- Elle définit des propriétés qui affectent la représentation visuelle.
- Les différentes zones d'une région sont basées sur la spécification du *Box-Model CSS3* (selon normalisation du W3C).
 - Elles définissent les notions *Margin*, *Border*, *Padding*, *Insets*, *Content*



Region - Structure visuelle [2]



- Comme la classe **Region** est la classe parente de tous les conteneurs et de tous les composants, ses propriétés sont héritées et peuvent s'appliquer à une grande variété d'éléments qui sont utilisés pour créer les interfaces.
- Parmi ces propriétés, on peut mentionner :
 - **border**
 - ⇒ Bordure autour de la région
 - **background**
 - ⇒ Couleur ou image d'arrière-plan de la région
 - **padding**
 - ⇒ Marge (espace) autour du contenu de la région
- L'utilisation et le codage des propriétés **border** et **background** sont expliqués plus en détail dans le chapitre suivant consacré aux conteneurs et *layout-panes* (à la section *Disposition des composants - Généralités*).

Disposition des composants [1]



- La qualité d'une interface graphique repose sur de nombreux éléments mais la disposition des composants dans la fenêtre figure certainement parmi les plus importants.
- Quand on parle de la **disposition (layout)** d'une interface, on s'intéresse plus particulièrement :
 - A la **taille** des composants
 - A la **position** des composants
 - ⇒ Position dans la fenêtre
 - ⇒ Position relative des éléments les uns par rapport aux autres
 - Aux **alignements** et **espacements** qui favorisent la structuration visuelle et influencent l'esthétique de l'interface
 - Aux **bordures** et aux **marges** (notamment autour des conteneurs)
 - Au **comportement dynamique** de l'interface lorsqu'on redimensionne la fenêtre, lorsqu'on déplace une barre de division (*splitpane*), lorsque le contenu change dynamiquement, etc.

Disposition des composants [2]



- Avec *JavaFX*, il est possible de dimensionner et de positionner les composants de manière absolue (en pixels, mm, etc.).
- Une **disposition avec des valeurs absolues** peut être utile et même nécessaire dans certaines situations particulières. Cependant, dans la plupart des cas, elle **présente de nombreux inconvénients**, car :
 - La **taille naturelle des composants** peut varier, en fonction
 - ⇒ De la langue choisie (libellés, boutons, menus, ...)
 - ⇒ De la configuration de la machine cible (paramètres de l'OS)
 - ⇒ Du *look & feel*, thème, *skin*, style (css) choisi par l'utilisateur
 - La **taille de la fenêtre** peut également varier
 - ⇒ Par le souhait de l'utilisateur
 - ⇒ Par obligation, pour s'adapter à la résolution de l'écran de la machine cible (pour afficher l'intégralité de l'interface)

Disposition des composants [3]



- Pour éviter ces inconvénients on préfère déléguer la disposition des composants à des **gestionnaires de disposition (*layout managers*)** qui sont associés à des conteneurs.
- L'idée principale est de définir des **règles de disposition** (des **contraintes**) que le gestionnaire se chargera de faire respecter en fonction du contexte spécifique de la machine cible.
- Avec *JavaFX*, les *layout managers* sont intégrés aux conteneurs (*layout-panes*) et ne sont pas manipulés directement par les programmeurs (seulement au travers des propriétés des conteneurs).
- C'est donc par le choix du *layout-pane* et en fonction des contraintes données que sont déterminées les règles de disposition.
- Plusieurs *layout-panes* sont prédéfinis dans *JavaFX*. Il est également possible de définir ses propres conteneurs avec des règles de disposition spécifiques, mais c'est rarement nécessaire.

Taille des composants

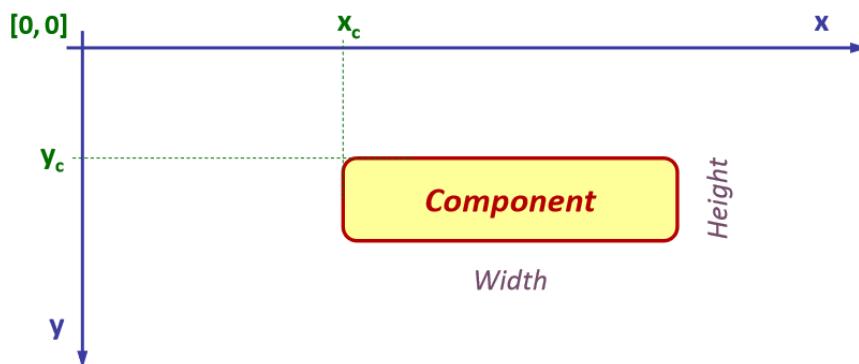


- Les composants (*nodes*) que l'on peut placer dans des conteneurs possèdent des propriétés qui peuvent être prises en compte lors du calcul de leur disposition.
 - `minWidth` : Largeur minimale souhaitée pour le composant
 - `prefWidth` : Largeur préférée (idéale) du composant
 - `maxWidth` : Largeur maximale souhaitée pour le composant
 - `minHeight` : Hauteur minimale souhaitée pour le composant
 - `prefHeight` : Hauteur préférée (idéale) du composant
 - `maxHeight` : Hauteur maximale souhaitée pour le composant
- L'effet de ces propriétés dépend naturellement du type de conteneur (*layout-pane*) utilisé et de ses règles spécifiques de positionnement et de dimensionnement.
 - Elles ne sont donc pas nécessairement prises en compte !

Système de coordonnées [1]



- Le **système de coordonnées** utilisé par *JavaFX* place l'origine dans le coin supérieur gauche du conteneur ou du composant considéré.
- Les valeurs de l'axe *x* croissent vers la droite.
- Les valeurs de l'axe *y* croissent vers le bas (traditionnel en infographie).
- La taille des composants est définie par leur largeur (*width*) et par leur hauteur (*height*).



Fenêtre principale [1]



- Par défaut, au lancement d'une application, la fenêtre principale (*primary stage*) est centrée sur l'écran.
- Différentes méthodes peuvent être invoquées sur l'objet **Stage** pour influencer la position et la taille de cette fenêtre :
 - `setX()` : Position en x du coin supérieur gauche
 - `setY()` : Position en y du coin supérieur gauche
 - `centerOnScreen()` : Centrage sur l'écran (par défaut)
 - `setMinWidth()` : Fixe la largeur minimale de la fenêtre
 - `setMinHeight()` : Fixe la hauteur minimale de la fenêtre
 - `setMaxWidth()` : Fixe la largeur maximale de la fenêtre
 - `setMaxHeight()` : Fixe la hauteur maximale de la fenêtre
 - `setResizable()` : Détermine si la fenêtre est redimensionnable
 - `sizeToScene()` : Adapte la taille de la fenêtre à la taille de la scène liée à cette fenêtre (utile si l'on change dynamiquement le contenu du graphe de scène)

Fenêtre principale [2]



- Autres méthodes de l'objet **Stage** :
 - `setTitle()` : Définit le titre de la fenêtre (affiché selon OS)
 - `setFullScreen()` : Place la fenêtre en mode plein-écran ou en mode standard (si paramètre `false`) (selon OS)
 - `getIcons().add()` : Définit l'icône dans la barre de titre
 - `setAlwaysOnTop()` : Place la fenêtre toujours au dessus des autres (généralement à éviter)
 - `setScene()` : Définit la scène (sa racine) qui est associée à la fenêtre
 - `show()` : Affiche la fenêtre à l'écran (et la scène qu'elle contient)
 - `showAndWait()` : Affiche la fenêtre à l'écran et attend que la fenêtre soit fermée pour retourner (méthode bloquante). Cette méthode n'est pas applicable à la fenêtre principale (*primary stage*).

Résumé des conteneurs [1]



Classe	Description
HBox, VBox	Place les composants horizontalement (sur une ligne) ou verticalement (dans une colonne).
FlowPane (horizontal)	Place les composants horizontalement sur une ligne et passe à la ligne suivante s'il n'y a plus assez de place dans le conteneur (<i>line-wrapping</i>).
FlowPane (vertical)	Place les composants verticalement (de haut en bas), en colonne et passe à la colonne suivante s'il n'y a plus assez de place dans le conteneur (<i>column-wrapping</i>).
TilePane (horizontal)	Place les composants dans une grille dont les cellules (les tuiles) ont toutes la même taille. Les composants sont ajoutés horizontalement, ligne par ligne.
TilePane (horizontal)	Place les composants dans une grille dont les cellules (les tuiles) ont toutes la même taille. Les composants sont ajoutés verticalement, colonne par colonne.

Classe	Description
BorderPane	Dispose de cinq emplacements pour placer les composants : <i>Top, Bottom, Left, Right, Center</i> .
AnchorPane	Place les composants en respectant une contrainte de distance par rapport à un ou plusieurs bords du conteneur.
StackPane	Place les composants les uns au dessus des autres (empilement). Le dernier ajouté est placé <i>au-dessus</i> des autres.
GridPane	Place les composants dans une grille potentiellement irrégulière (par défaut, la taille des lignes et des colonnes est déterminée par le plus grand composant qui y est placé). Les composants sont ajoutés en donnant l'indice de la colonne et de la ligne (la numérotation commence à zéro). La zone d'un composant n'est pas limitée à une seule cellule, elle peut s'étendre sur plusieurs colonnes et plusieurs lignes (<i>spanning</i>). Un composant peut s'agrandir pour occuper toute sa zone.

Composants – Controls [1]

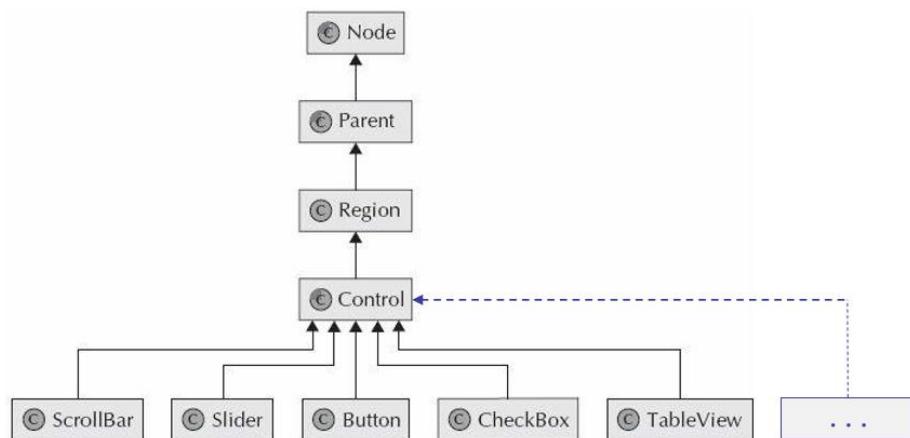


- La librairie JavaFX offre un **ensemble de composants** (kit de développement) pour créer les interfaces utilisateurs graphiques.
- Ces **composants d'interface** sont fréquemment nommés **controls** dans la documentation en anglais (parfois **widglets**).
- Dans ce cours, nous utiliserons généralement le terme **composant** pour parler des éléments qui servent à afficher des informations ou permettre à l'utilisateur d'interagir avec l'application.
 - Libellés, icônes, boutons, champs-texte, menus, cases à cocher, etc.
- Bien qu'ils constituent les deux des nœuds (*node*) du graphe de scène, les **composants** sont à distinguer des **conteneurs** (*layout-panes*) qui servent à disposer les composants et qui ne sont pas directement visibles dans l'interface (les bordures et les couleurs d'arrière-plan permettent cependant de révéler leur présence).

Composants – Controls [2]



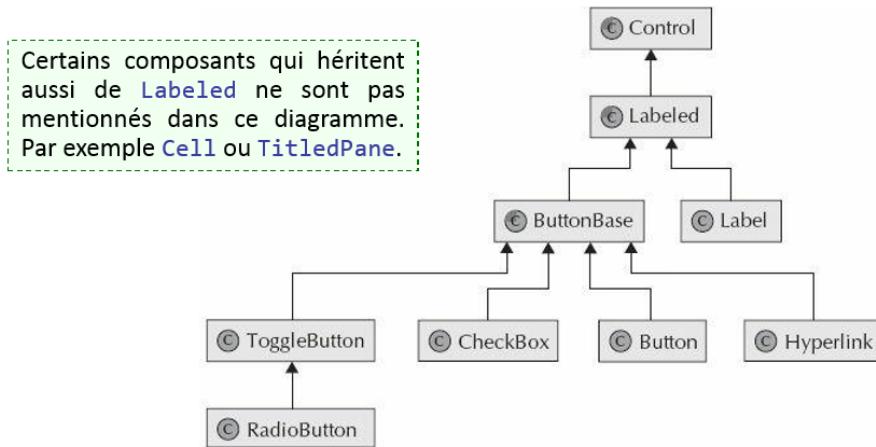
- Les composants ont tous pour classe parente **Control** qui est une sous-classe de **Node**.
- Une version simplifiée des dépendances entre les différentes classes est illustrée par le diagramme suivant :



Composants avec libellés [1]



- De nombreux composants affichent et gèrent des textes (libellés, boutons, cases à cocher, etc.).
- Les comportements communs de ces composants sont gérés par la classe parente **Labeled**.



Composants avec libellés [2]



- Les textes de ces composants peuvent être accompagnés d'un autre composant, généralement un graphique, une image ou une icône.
- Quelques propriétés communes aux composants **Labeled** :

text	Texte affiché (String).
font	Police de caractères (famille, style, taille, ...), type Font .
textFill	Couleur du texte, uniforme ou avec gradient (type Paint).
underline	Indique si le texte doit être souligné (type Boolean).
alignment	Alignement général du texte (et du graphique éventuel) dans la zone (type Pos). Valable seulement si texte sur une seule ligne.
wrapText	Booléen qui définit si le texte passe à la ligne suivante lorsqu'il atteint la limite de la zone. Le caractère '\n' peut également être inséré pour forcer un retour à la ligne (inconditionnel).
textAlignment	Alignement des lignes si le texte est multiligne. Type énuméré TextAlignment (LEFT , RIGHT , CENTER , JUSTIFY).
lineSpacing	Espacement des lignes pour les textes multilignes. Type Double .

Label [1]

JavaFX



- Le composant **Label** représente un libellé (= un texte non éditable).
- Les constructeurs permettent de définir le contenu du texte et de l'éventuel composant additionnel (*graphic*).
 - `new Label("Hello");`
 - `new Label("Warning", warningIcon);`
- L'essentiel des fonctionnalités sont héritées de **Labeled**. Une seule propriété additionnelle se trouve dans **Label** :
 - `setLabelFor` : Permet de définir un (autre) composant (**Node**) auquel le libellé est associé (utile pour définir un mnémonique).

Remarque : Les objets de type **Text** possèdent certaines similitudes avec les composants de type **Label** mais **Text** n'est pas une sous-classe de **Control**. Cette classe fait partie de la famille des graphiques (sous-classe de **Shape**) et possède donc des propriétés et des fonctionnalités un peu différentes.

Label [2]

JavaFX



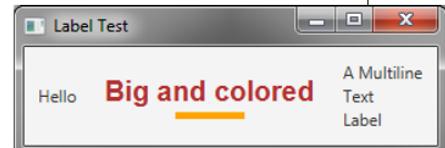
```
private HBox root = new HBox(20);

private Label lblA = new Label("Hello");
private Label lblB = new Label("Big and colored");
private Label lblC = new Label("A Multiline\nText\nLabel");

primaryStage.setTitle("Label Test");
root.setAlignment(Pos.CENTER);
root.setPadding(new Insets(10));
root.getChildren().add(lblA);

lblB.setFont(Font.font("SansSerif", FontWeight.BOLD, 20));
lblB.setTextFill(Color.rgb(180, 50, 50));
lblB.setGraphic(new Rectangle(50, 5, Color.ORANGE));
lblB.setContentDisplay(ContentDisplay.BOTTOM);
root.getChildren().add(lblB);
root.getChildren().add(lblC);

primaryStage.setScene(new Scene(root));
primaryStage.show();
```



Button [1]



- Le composant **Button** représente un bouton permettant à l'utilisateur de déclencher une action.
- La classe parente **ButtonBase** rassemble les propriétés communes à différents composants qui se comportent comme des boutons : **Button**, **CheckBox**, **Hyperlink**, **MenuButton**, **ToggleButton**.
- Les constructeurs permettent de définir le contenu du texte et de l'éventuel composant additionnel (*graphic*).
 - `new Button("Ok");`
 - `new Button("Save", saveIcon);`
- Par héritage, toutes les propriétés qui ont été mentionnées pour les composants avec libellés (sous-classes de **Labeled**) sont naturellement applicables pour le composant **Button**.

Button [2]



- Quelques propriétés du composant **Button** :

<code>armed</code>	Booléen qui indique si le bouton est " <i>armé</i> " et prêt à déclencher une action (par exemple souris placée sur le bouton et touche gauche pressée).
<code>onAction</code>	Détermine l'événement à générer lorsque l'action du bouton est déclenchée (par ex. lorsque la touche de la souris a été relâchée). Type <code>EventHandler<ActionEvent></code> .
<code>cancelButton</code>	Booléen qui indique si le bouton est un bouton <i>Cancel</i> c'est-à-dire que l'action du bouton doit être déclenchée lorsque l'utilisateur presse sur la touche <i>Escape</i> (<code>VK_ESC</code>). Cette propriété ne doit être appliquée qu'à un seul bouton de l'interface.
<code>defaultButton</code>	Booléen qui indique si le bouton est un bouton par défaut c'est-à-dire que l'action du bouton doit être déclenchée lorsque l'utilisateur presse sur la touche <i>Enter</i> (<code>VK_ENTER</code>). Cette propriété ne doit être appliquée qu'à un seul bouton de l'interface.

Saisie de textes [1]



- La classe abstraite **TextInputControl** est la classe parente de différents composants qui permettent à l'utilisateur de saisir des textes. Il s'agit notamment des composants d'interface **TextField**, **PasswordField** et **TextArea**
- La classe **TextInputControl** définit les propriétés de base et les fonctionnalités communes aux composants qui offrent une saisie de texte et notamment :
 - La sélection de texte
 - L'édition de texte
 - La gestion du curseur à l'intérieur du texte (*caret*)
 - Le formatage du texte

Saisie de textes [2]



- Quelques propriétés de la classe **TextInputControl** :

text	Le texte contenu dans le composant (String).
editable	Le texte peut être édité par l'utilisateur (Boolean).
font	Police de caractères du texte (Font).
length	Longueur du texte (Integer).
caretPosition	Position courante du curseur / point d'insertion (<i>caret</i>).
promptText	Texte affiché si aucun texte n'a été défini ou saisi par l'utilisateur (String). Ce texte n'est pas affiché lorsque le composant possède le focus (avec le curseur qui clignote dans le champ). Ce texte peut éventuellement remplacer un libellé ou une bulle d'aide pour le champ texte.
textFormatter	Formateur de texte associé au composant (TextFormatter<?>).
selectedText	Texte sélectionné (String).
selection	Indices (de...à) de la zone sélectionnée (IndexRange).
anchor	Point d'ancrage (début) de la sélection (Integer).

Saisie de textes [3]



- Quelques méthodes de la classe **TextInputControl** :

<code>clear()</code>	Efface le texte (vide le champ).
<code>copy/cut/paste()</code>	Transfert du texte dans ou depuis le <i>clipboard</i> .
<code>positionCaret()</code>	Positionne le curseur à une position donnée.
<code>forward()</code> <code>backward()</code>	Déplace d'un caractère le curseur (<i>caret</i>).
<code>nextWord()</code>	Déplace le curseur (<i>caret</i>) au début du prochain mot.
<code>insertText()</code>	Insère une chaîne de caractères dans le texte.
<code>appendText()</code>	Ajoute une chaîne de caractères à la fin du texte.
<code>deleteText()</code>	Supprime une partie du texte (de...à).
<code>deleteNextChar()</code>	Efface le prochain caractère.
<code>replaceText()</code>	Remplace une partie du texte par un autre.
<code>selectAll()</code>	Sélectionne l'ensemble du texte.
<code>deselect()</code>	Annule la sélection courante du texte.

TextField [1]



- Le composant **TextField** représente un champ texte d'une seule ligne qui est éditable par défaut mais qui peut également être utilisé pour afficher du texte.
- Le composant n'intègre pas de libellé. Il faut utiliser un composant de type **Label** si l'on veut lui associer un libellé.
- En plus des propriétés héritées (notamment de **TextInputControl**), le composant **TextField** possède les propriétés suivantes :

<code>alignment</code>	Alignement du texte dans le champ (Pos).
<code>prefColumnCount</code>	Nombre de colonnes du champ texte; permet de déterminer la largeur préférée du composant. La valeur par défaut est définie par la constante DEFAULT_PREF_COLUMN_COUNT (12).
<code>onAction</code>	Détermine l'événement à générer lorsque l'action du champ texte est déclenchée, en général lorsque l'utilisateur presse sur la touche <i>Enter</i> (EventHandler<ActionEvent>).

TextFormatter [1]



- On peut associer un formateur de texte à tous les composants qui héritent de `TextInputControl` (propriété `TextFormatter`).
- Ce formateur est un composant de type `TextFormatter<V>` qui permet de définir :
 - Un **convertisseur** permettant de convertir le texte du composant en une valeur d'un autre type (par exemple un type numérique, `int`, `double`, ...).
 - Un **filtre** permettant d'intercepter et de modifier les caractères saisis par l'utilisateur pendant l'édition du texte (n'accepter que les chiffres par ex.).
- Le formateur peut définir un filtre ou un convertisseur ou les deux.
- Le filtre et le convertisseur sont transmis dans le constructeur du formateur qui possède les surcharges suivantes :

```
TextFormatter(StringConverter<V> valueConverter)
TextFormatter(StringConverter<V> valueConverter, V defaultValue)
TextFormatter(UnaryOperator<TextFormatter.Change> filter)
TextFormatter(StringConverter<V> valueConverter, V defaultValue,
             UnaryOperator<TextFormatter.Change> filter)
```

TextFormatter [2]



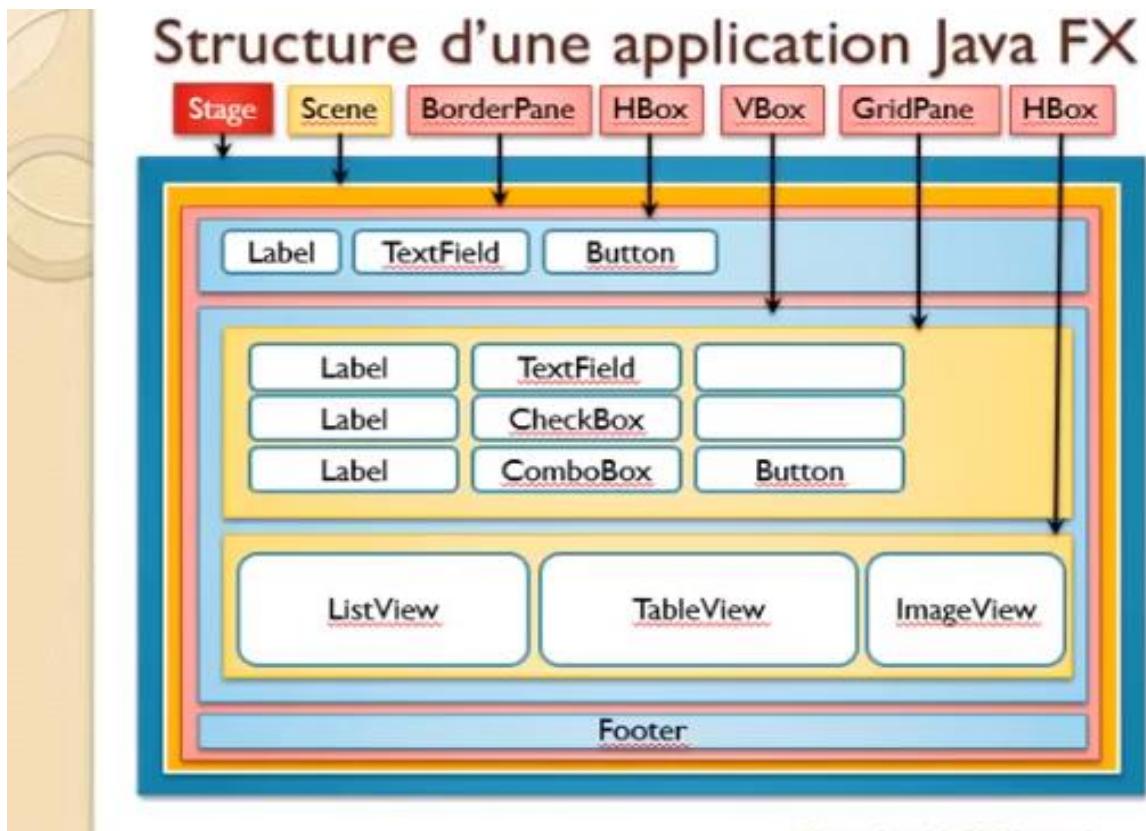
- Le paramètre générique `V` du formateur (`TextFormatter<V>`) définit le type de la propriété **value**. On ne peut utiliser cette propriété que si l'on a défini un convertisseur dans le formateur.
- Le **convertisseur** est un objet de type `StringConverter<V>` qui doit implémenter les méthodes de conversions entre les propriétés `text (String)` et `value (V)` :
 - `fromString() : text → value`
 - `toString() : value → text`
- Il existe des implémentations prédéfinies de convertisseurs pour certains types courants :
 - `BooleanStringConverter`, `DoubleStringConverter`, `IntegerStringConverter`, `NumberStringConverter`, `DateTimeStringConverter`, ...
- Si l'on souhaite gérer le format d'affichage et/ou traiter certaines erreurs, il est préférable de redéfinir les méthodes de conversion.

TextFormatter [3]



- Exemple de convertisseur associé à un champ texte.

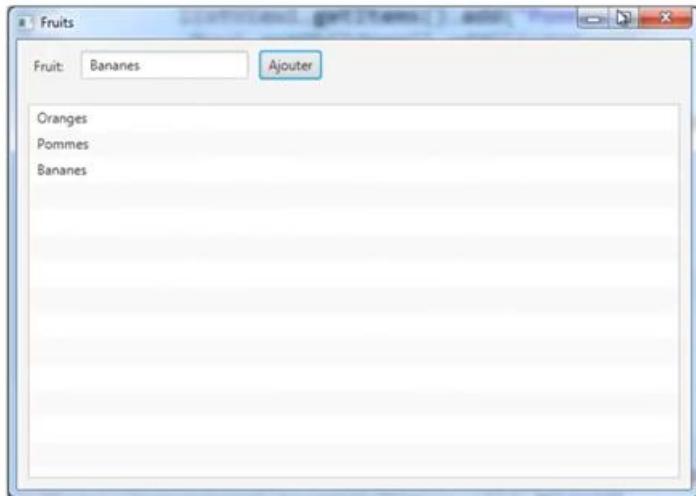
```
//----- TextField Formatter -----
TextFormatter<Double> tFmt = new TextFormatter<Double>(
    new StringConverter<Double>() {
        @Override //--- Convert text (String) to Double (NaN if not possible)
        public Double fromString(String text) {
            try {
                return Double.parseDouble(text);
            }
            catch (NumberFormatException e) {
                return Double.NaN;
            }
        }
        @Override //--- Convert Double to String and format (2 decimals)
        public String toString(Double value) {
            return String.format("%.2f", value);
        }
    }
);
tfdPrice.setTextFormatter(tFmt);
```



TP1 : Créer l'interface ci-dessous :



Premier Exemple JavaFX



Utilisation de CSS pour personnaliser le design des composants

- Java FX donne la possibilité de personnaliser les attributs d'affichage des composants graphique en utilisant des feuilles de styles CSS.

```
Button buttonAdd=new Button("Ajouter");
Button buttonSupp=new Button("Supp");
buttonSupp.getStyleClass().add("myButton");
scene.getStylesheets().add(getClass().getResource("myStyle.css").toString());
```

myStyle.css

```
.button{
-fx-background-color: orange;
-fx-border-color:blue;
-fx-border-width:3;

}
.myButton{
-fx-background-color: yellow;
-fx-border-color:red;
-fx-border-width:2;
```



Méthode basée sur FXML

- Java FX offre la possibilité de déclarer la structure des composants de l'interface graphique dans une fichier XML.
- Le Fichier FXML représente la vue de l'application
- Au démarrage de l'application, le fichier XML est lu, les composants sont instanciés et l'interface graphique s'affiche.
- Pour programmer les réponses aux évènements produits dans les composants, on crée un Contrôleur associé à la vue FXML.



TP1 : Créer la même interface en utilisant SceneBuilder

Utilitaires

1) Chargement d'une page FXML

Parent root =

FXMLLoader.load(getClass().getResource(view));

View : représente url de la view

2) Récupérer Objet Windows à partir d'un Control

control.getScene().getWindow() ;

3) Effacer un layout

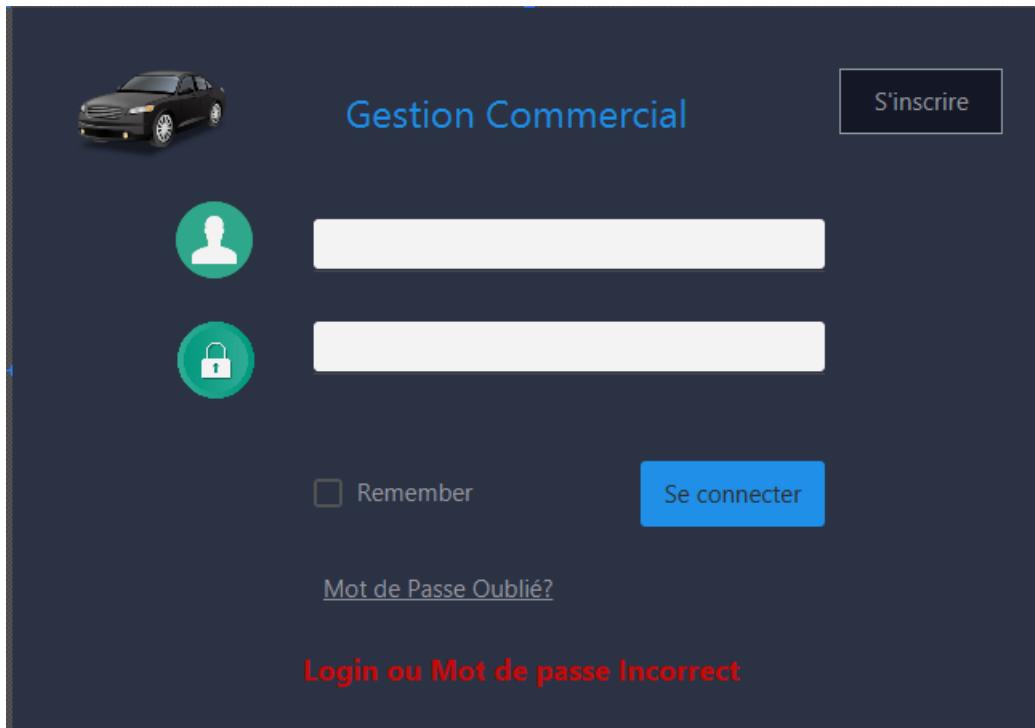
layout.getChildren().clear();

4) Ajouter un layout dans un autre layout

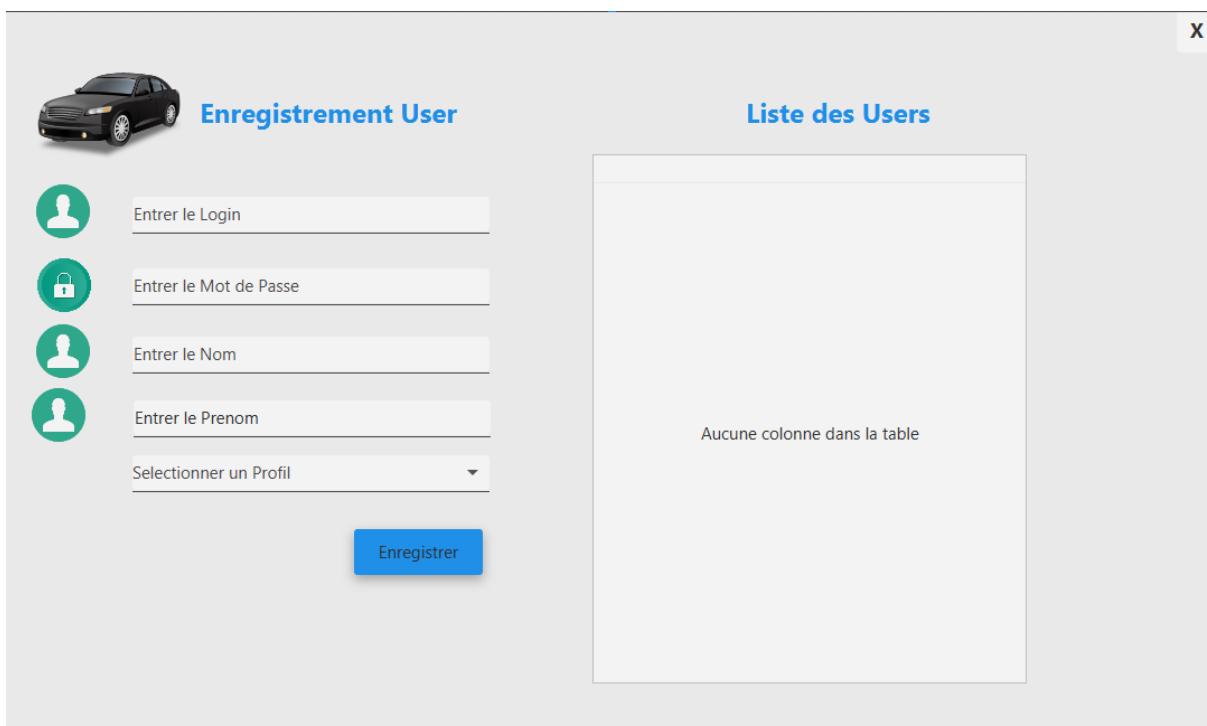
anchor_content.getChildren().add((Node)node);

TP2 : Créer les interfaces ci-dessous

vlogin.fxml



vinscription.fxml



varticle.fxml

Enregistrement Article

Reference _____

Libelle _____

Stock _____

Prix _____

Catégorie _____ ▾

Enregistrer

Gestion Commercial

ID	Reference	Libelle	Stock	Prix	Catégorie
Aucun contenu dans la table					

vcommande.fxml

Enregistrement Client

Nom
Prenom
Telephone
Adresse

Enregistrer

Numero X

Date ADD

Telephone Search

Nom Prenom Adresse

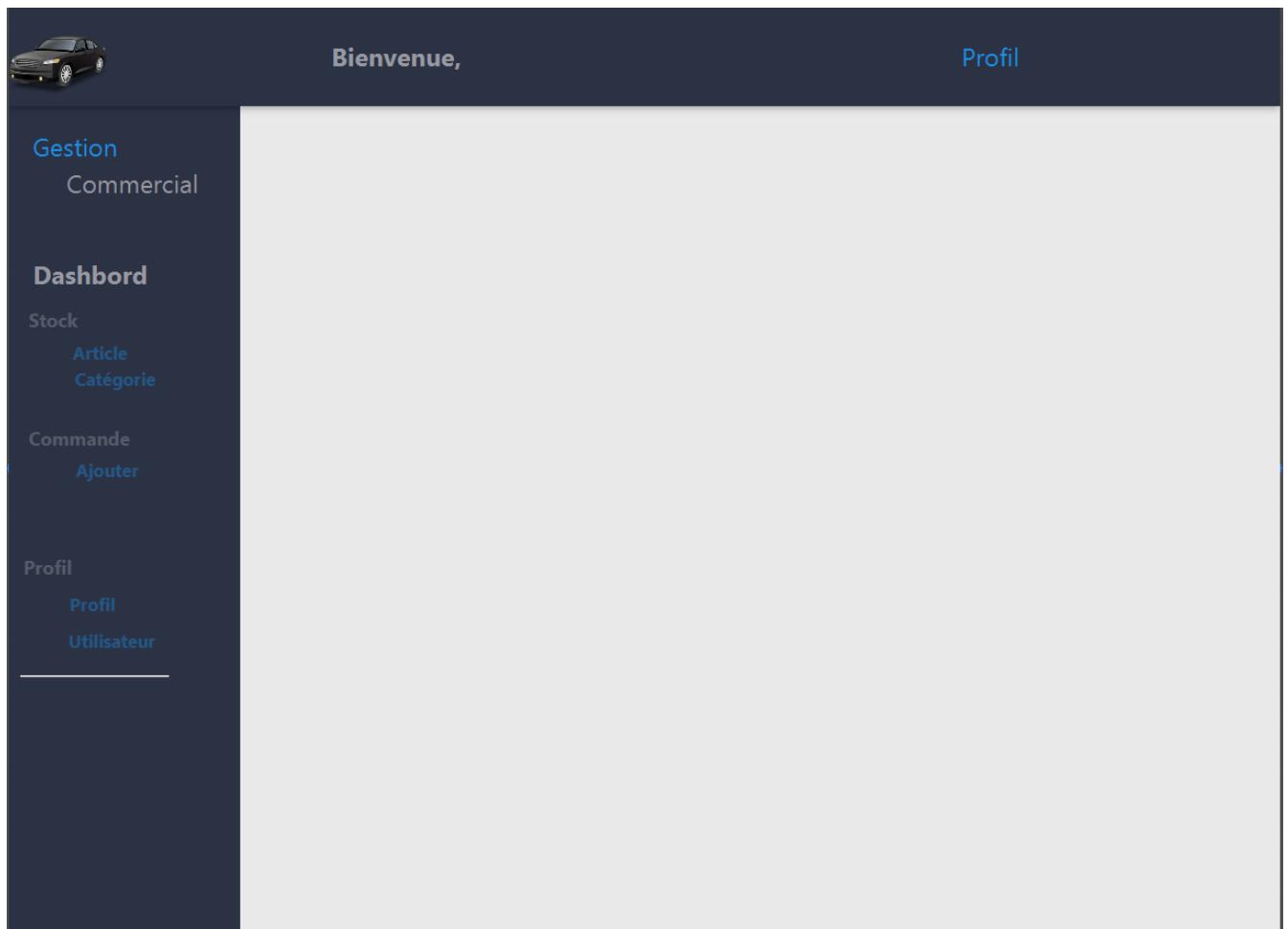
Reference Search

Libelle Qte Stock Qte Commandée ADD

ID	Libelle	Qte Comd	Prix	Montant
Aucun contenu dans la table				

Total:

vdashboard.fxml



Couleurs

#2D3447 : Background
#2D3447 : Background layout
#a0a2ab: Police
#2196f3 : Background button
#151928 : Background button
#33b86c : Vert

#ef5350 : Rouge #ffd740 : warning

Utilisation de Quelques Composants FXML

❖ Alert

- Alert d'Information

```
Alert alert=new Alert(Alert.AlertType.INFORMATION);
alert.setContentText("Utilisateur Enregistré avec succès");
```

- Alert de Confirmation

```
Alert alert=new Alert(Alert.AlertType.CONFIRMATION);
alert.setContentText("Voulez Supprimer cette Utilisateur ?");
Optional<ButtonType> result = alert.showAndWait();
if (result.get() == ButtonType.OK){
    listUser.remove(pos);
    Fabrique.getUserController().delete(user);
}
```

❖ TableView

- Initialisation

```
private TableView<> tblvUser;
private TableColumn<User, String> tblcId;
```

```
//Indiquer la colonne à afficher
tblcId.setCellValueFactory(new PropertyValueFactory<>("id"));
tblvUser.setItems(ObservableList<User>);
```

- Sélection d'une ligne

```
User user=tblvUser.getSelectionModel().getSelectedItem();
```

- Sélection de la position de la Ligne sélectionnée

```
int pos=tblvUser.getSelectionModel().getSelectedIndex();
```

❖ JFXComboBox

- Initialisation

```
JFXComboBox<Profil> cbxProfil;
cbxProfil.setItems(ObservableList<Profil>);
```

- Récupérer l'élément selectionné

```
Profil p=cbxProfil.getSelectionModel().getSelectedItem()
```

Cours 2 : JDBC (JAVA DATABASE CONNECTIVITY)

Introduction

JDBC est une API Java (ensemble de classes et d'interfaces défini par SUN et les acteurs du domaine des BD) permettant d'accéder aux bases de données à l'aide du langage Java via des requêtes SQL. Cette API permet d'atteindre de manière quasi-transparente des bases **Sybase, Oracle, MySQL, ...** avec le même programme Java JDBC.

En fait cette API est une spécification de ce que doit implanter un constructeur de BD pour que celle-ci soit interrogable par JDBC. De ce fait dans la programmation JDBC on utilise essentiellement des références d'interface (**Connection, Statement, ResultSet, ...**).

Sun et les constructeurs de BD se sont chargés de fournir (vendre ou donner) des classes qui implémentent les interfaces précitées qui permettent de soumettre des requêtes SQL et de récupérer le résultat.

Exemple MySQL fournit une classe qui, lorsqu'on écrit

Statement stmt = conn.createStatement(); retourne un objet concret (de classe class **MysqlStatement** implements Statement par exemple) qui est repéré par la référence stmt de l'interface Statement.

I) Pilotes (drivers)

L'ensemble des classes qui implémentent les interfaces spécifiées par JDBC pour un gestionnaire de bases de données particulier est appelé un **pilote JDBC**. Les protocoles d'accès aux BD étant propriétaires il y a donc plusieurs drivers pour atteindre diverses BD.

Parmi les interfaces, l'une d'entre elles, l'interface Driver, décrit ce que doit faire tout objet d'une classe qui implémente l'essai de connexion à une base de données.

Pilote MySQL : com.mysql.jdbc.Driver

Entre autre, un tel objet doit obligatoirement s'enregistrer auprès du DriverManager et retourner en cas de succès un objet d'une classe qui implémente l'interface Connection.

II) Structure d'un programme JDBC

Un code JDBC est de la forme :

1) Recherche et chargement du driver approprié à la BD.

```
Class.forName("com.mysql.jdbc.Driver");
```

2) Etablissement de la connexion à la base de données.

Connection cnx=

```
DriverManager.getConnection("jdbc:mysql://localhost:3306/dbname",
                           "user","password");
```

3) Construction de la requête SQL

On utilise l'un des trois interfaces **Statement**,**PreparedStatement**,
CallableStatement pour exécuter une requête SQL.

Les instructions **PreparedStatement** sont précompilées et permettent d'exécuter des requêtes paramétrées.

On utilise **CallableStatement** pour lancer une procédure du SGBD.

Exemple

```
Statement stmt = cnx.createStatement();
```

4) Envoi de cette requête et récupération des réponses

Syntaxiquement en Java on utilise la méthode **executeQuery(...)** si la requête est une requête SELECT et la méthode **executeUpdate(...)** si la requête est une requête d'action ou une SQL DDL.

Exemple

❖ Avec Statement

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c ... FROM ...
                                  WHERE ...");
```

```
int = stmt. executeUpdate (Insert into ...);
```

❖ Avec Prepared Statement

```
PreparedStatement pSmt = cnx.prepareStatement("SELECT
                                             A,B FROM table where id= ?.....");
```

Puis on utilise les méthodes **pSmt.setType(numéroDeLArgument, valeur)** pour positionner chacun des arguments. Les numéros commencent à 1 dans l'ordre d'apparition dans la requête.

Apres on exécute la requete

```
ResultSet rs = pSmt.executeQuery();
```

NB : on peut utiliser

```
pstm=cnx.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);  
lorsqu'on veut récupérer le dernier id insérer.
```

5) Parcours des réponses

```
while (rs.next()) {  
    ...  
    // traitement  
}  
rs.close () ; //Fermer le ResultSet
```

- **rs.next()** :récupère une ligne de la requête
- **rs.getType(numéroDelaColonne)** : récupère une colonne d'une ligne de la requête.

6) Fermer la connexion

```
cnx.close () ; //pour fermer la connexion
```

Exemple De Programme :

```
Class.forName("com.mysql.jdbc.Driver");
```

Connection cnx=

```
DriverManager.getConnection ("jdbc:mysql://localhost:3306/gescom", "root","", "");
```

a) Requête Select

```
String sql= "select libelle,pu from article where reference= ?" );  
PreparedStatement pSmt = cnx.prepareStatement(sql) ;  
pSmt.setString(1, "xxx") ;  
ResulSet rs=pSmt.executeQuery() ;  
if(rs.next()){  
    Article art=new Article()  
    Art.setLibelle(rs.getString(1)) ;  
    Art.setDouble(rs.getDouble(2)) ;  
}  
rs.close() ;
```

b) Requête insert

```
String sql= "insert into (libelle,pu) values( ?, ?)" ;  
PreparedStatement pSmt = cnx.prepareStatement(sql) ;  
pSmt.setString(1, "Lait") ;  
pSmt.setDouble(2, 1000) ;  
int id=pSmt.executeUpdate() ;  
rs.close() ;
```

c) Requête insert qui retourne le dernier Id

```
String sql= "insert into (libelle,pu) values( ?, ?)" ;
PreparedStatement pSmt = cnx.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS) ;
pSmt.setString(1, "Lait") ;
pSmt.setDouble(2, 1000) ;
ResulSet rs =pSmt.executeUpdate() ;
int id ;
if(rs.next()){
    id=rs.getInt() ;
}
rs.close() ;
```

III) Classes De Connexion

Config.java

```
public class Config {
    protected static final String dbhost="localhost";
    protected static final String dbport="3306";
    protected static final String dbuser="root";
    protected static final String dbpass="";
    protected static final String dbname="gescom";

}
```

DaoMysql.java

```
public class DbConnection extends Config {
    private Connection cnx;
    private PreparedStatement pstm;
    private int ok;
    private ResultSet rs;

    public void getConnection()
    {
        cnx=null;
        try{
            Class.forName("com.mysql.jdbc.Driver");

            cnx=DriverManager.getConnection("jdbc:mysql://" +Config.dbhost+ ":" +Config.dbport+ "/" +Config.dbname,
                Config.dbuser,Config.dbpass);
        }catch(ClassNotFoundException | SQLException e)
        {
            e.printStackTrace();
        }
    }

    public void initPS(String sql)
    {
        getConnection();
        try{
            if(sql.toLowerCase().startsWith("insert"))
            {
                pstm=cnx.prepareStatement(sql,
                    Statement.RETURN_GENERATED_KEYS );
            }
            else{
                pstm=cnx.prepareStatement(sql);
            }
        }catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```
}

public int executeMaj()
{
    try {
        ok = pstm.executeUpdate();

    } catch (Exception e) {

        e.printStackTrace();

    }
    return ok;
}
public ResultSet executeSelect()
{
    try {

        rs=pstm.executeQuery();

    } catch (Exception e) {
        e.printStackTrace();

    }
    return rs;
}
public PreparedStatement getPstm()
{
    return this.pstm;

}
```

```
public void CloseConnection(){
    try{
        if(cnx!=null){
            cnx.close();
        }
    }catch(Exception ex){
        ex.printStackTrace();
    }
}
```