

# *Interfaces graphiques avec JavaFX*

1 - Installation JavaFX et <i>SceneBuilder</i>	1
2 - Introduction à JavaFX	7
3 - Les contrôles de JavaFX	17
4 - Les <i>layouts</i> ou gestionnaires de positionnement	31
5 - Autres notions : boîte de dialogues et CSS	37
6 - Utilisation de <i>SceneBuilder</i>	44

# Installation de JavaFX et de SceneBuilder Eclipse

## ETAPE 1 : Télécharger et installer JavaFX

Remarque : avec les versions 8 à 10 de Java, JavaFX était inclus dans le JRE de Java. Depuis Java 11, ce n'est plus le cas.

JavaFX est géré par une communauté Open Source **openjfx.io**. Il faut donc aller sur le site de **openjfx.io** pour télécharger JavaFX. Une fois sur le site :

- 1) cliquer sur *Download*



- 2) une redirection est faite vers le site de téléchargement. Dans la rubrique *Download* il faut rechercher la version qui correspond à l'ordinateur sur lequel on souhaite effectuer l'installation (probablement le système Windows avec une architecture 64 bits). Cliquer sur le bouton vert « *Download* »

OS	Version	Architecture	Type	Download
Windows	20	x64	SDK	<a href="#">Download</a> [SHA256]

- 3) Il faut maintenant décompresser l'archive téléchargée. Pour se faire, sur le disque C (ou autre éventuellement) à côté du dossier contenant le SDK de Java (probablement un dossier nommé « Java » situé dans le dossier « Programmes »), créer un dossier nommé **JavaFX** et décompresser dans celui-ci l'archive téléchargée.

## ETAPE 2 : Ajouter à Eclipse le plugin e(fx)clipse

Pour travailler avec JavaFX dans l'environnement Eclipse, il est nécessaire d'ajouter à ce dernier un plugin. Ce plugin se nomme **e(fx)clipse**. Il faut connaître la dernière version de ce plugin, actuellement la 3.8.0. Pour obtenir ce numéro, on peut aller sur le site d'Eclipse :

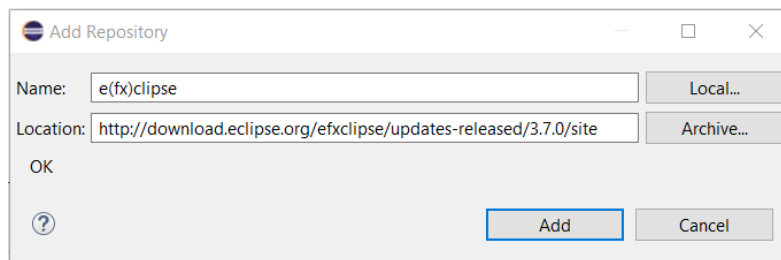
<https://projects.eclipse.org/projects/technology.efxclipse>

- 1) Lancer Eclipse et faire : Help -> Install New Software...

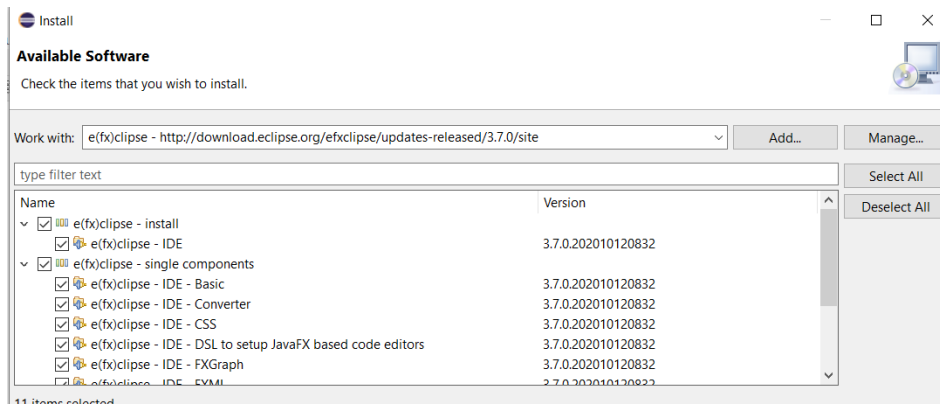


2) Cliquer sur « Add » pour ajouter un nouveau dépôt à partir duquel il sera possible de télécharger des plugins Eclipse.

Dans la fenêtre qui s'ouvre renseigner le nom que l'on souhaite attribuer au dépôt (par exemple e(fx)clipse) puis l'URL du dépôt qui comporte le numéro de la version que l'on souhaite installer (voir copie d'écran ci-dessous), ici la 3.8.0. Cliquer sur « Add ».



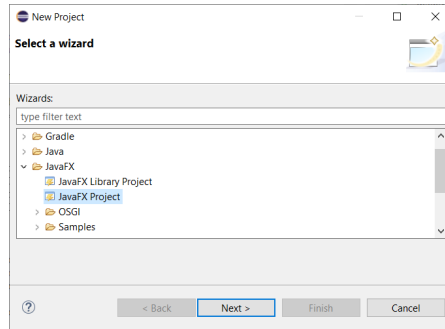
3) La fenêtre de départ affiche maintenant les composants à installer, les 2 principaux et des sous-composants. Cocher les 2 composants principaux et cliquer sur « Next » à 2 reprises, puis accepter les termes de la licence avant de cliquer sur « Finish ».



Une fois l'installation terminée (l'installation dure quelques secondes ou minutes, voir en bas à droite le message « Installing software ... »), Eclipse va vous demander de relancer l'environnement, ce qu'il faut faire.

## ETAPE 3 : Créer un premier projet JavaFX

1) Faire `File -> New -> Project...` Puis choisir de créer un projet JavaFX (dérouler en dessous JavaFX).



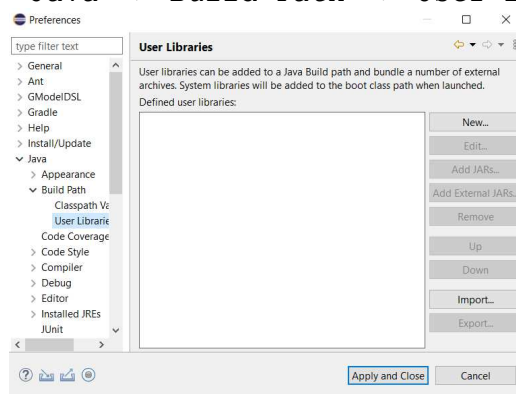
Cliquer sur « *Next* ».

Dans la fenêtre suivante, renseigner le nom du projet, puis cliquer sur « *Finish* ».

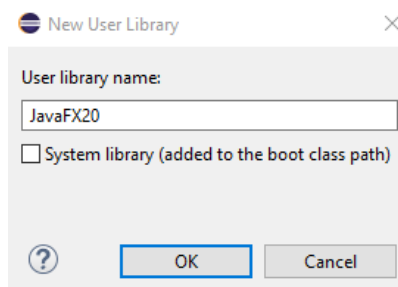
On constate que le projet est pour l'instant en erreur : le package **javafx** n'est pas trouvé (voir le code de la classe *Main* pour faire le constat de l'erreur).

2) Nous devons intégrer les bibliothèques **JavaFX** dans Eclipse. Pour ce faire :

`Window -> Preferences -> Java -> Build Path -> User Libraries`



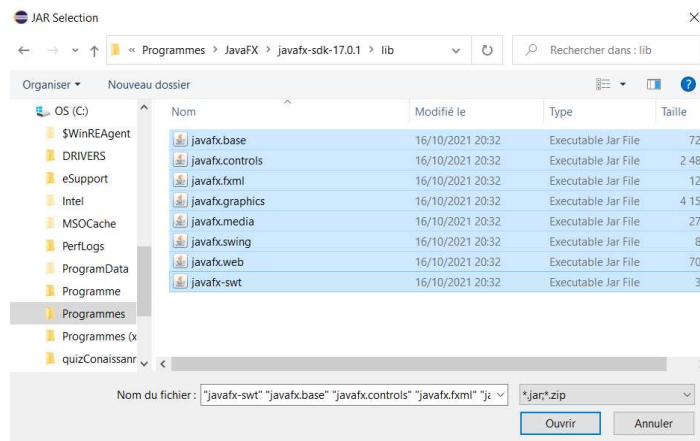
3) Cliquer sur “*New*”. Renseigner un nom de bibliothèque à créer, JavaFX20 sur la copie d'écran par exemple. Cliquer sur « *OK* »



4) Une fois revenu sur la fenêtre précédente, cliquer sur « *Add External JARs* ». Parcourir l'arborescence pour aller dans le dossier d'installation de JavaFX, aller jusqu'au sous-dossier **lib** et sélectionner tous les .jar présents dans ce dossier, puis cliquer sur « *Ouvrir* ».

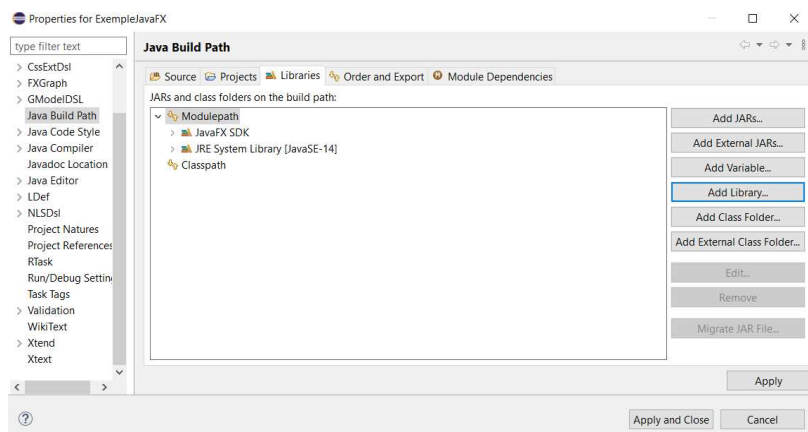
**Remarque** : sur les ordinateurs de l'IUT, le dossier est **D:/Eclipse IDE/javafx-sdk-18-01**

=> voir copie d'écran ci-après



5) Sur la fenêtre précédente, cliquer sur « *Apply and close* ».

6) Il faut maintenant ajouter cette bibliothèque au projet. Pour ce faire : clic droit sur le projet (dans le **Package Explorer**), puis Build Path -> Configure Build Path



Aller sur l'onglet « **Librairies** » et cliquer sur « **Add Library...** ». Dans la fenêtre qui s'ouvre choisir « **User Library** », cliquer sur « **Next** ». Cocher ensuite la bibliothèque JavaFX20, puis cliquer sur « **Apply and close** ».

7) Le projet n'est plus en erreur. Vous allez maintenant lancer le projet (bouton vert « **Run** ») et vous allez peut être voir apparaître l'erreur suivante :

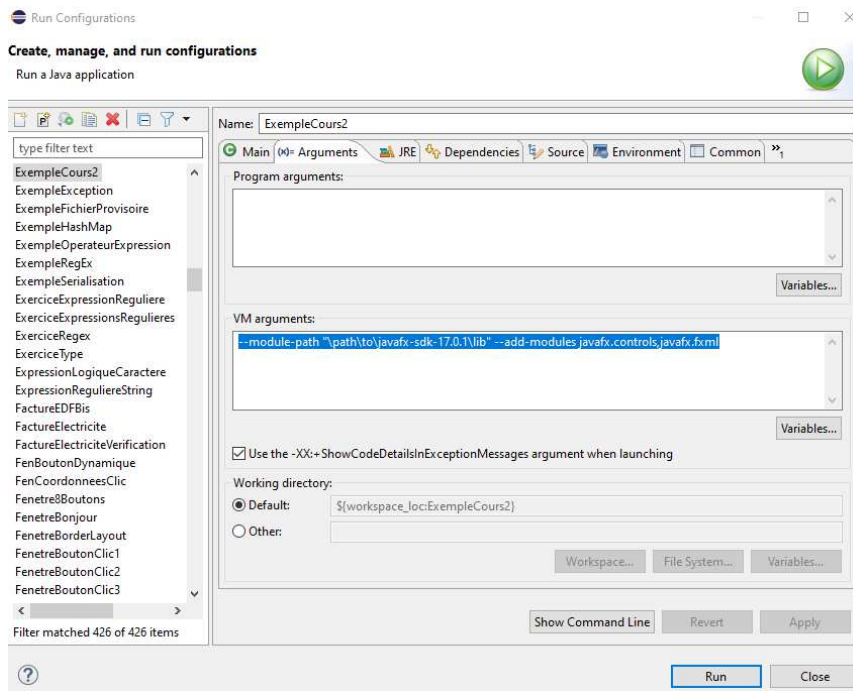
Error: Could not find or load main class application.Main  
Caused by: java.lang.NoClassDefFoundError: javafx/application/Application

Pour la corriger, il faut modifier les paramètres d'exécution de la machine virtuelle. Pour ce faire : clic droit sur le nom du projet (dans le **Package Explorer**), puis Run as -> Run configurations...

Dans la liste des classes affichées à gauche de la fenêtre, sélectionnez la classe à exécuter (normalement elle est déjà sélectionnée). Puis aller sur l'onglet « **Arguments** ». Dans la rubrique « **VM arguments** », ajouter la ligne suivante :

```
--module-path "\path\to\javafx-sdk-18.0.1\lib" --add-modules javafx.controls,javafx.fxml
```

=> voir copie d'écran ci-après (attention : adapter la version de javafx au contexte)



Puis cliquer sur “**Run**”. L’application JavaFx va maintenant se lancer.

## ETAPE 4 : Installer SceneBuilder

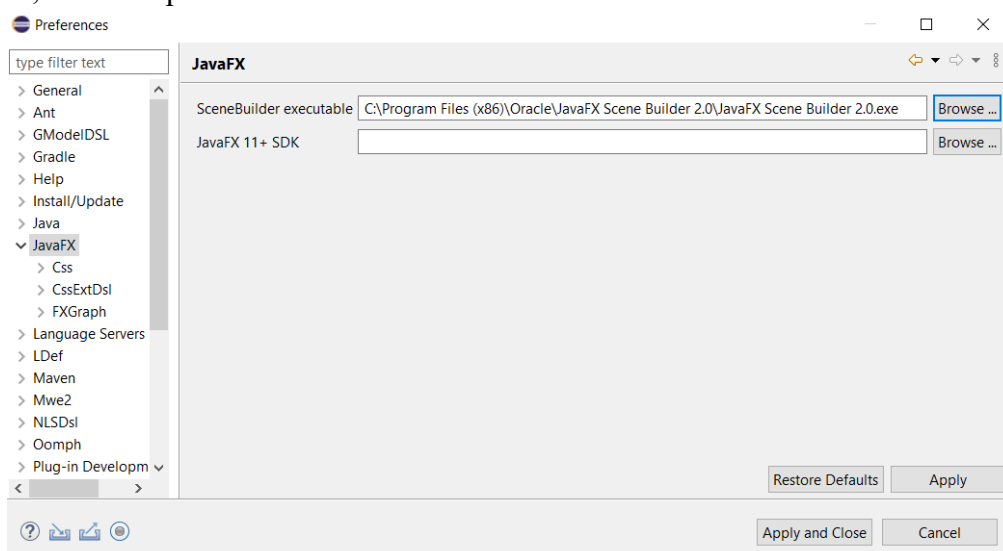
**SceneBuilder** est un outil interactif pour concevoir des interfaces graphiques pour JavaFX. Il évite au programmeur de coder directement la description de la partie statique de l’interface. Pour ce faire, **SceneBuilder** génère automatiquement du code dans le langage *fxml*.

Aller sur le site d’Oracle pour télécharger **SceneBuilder**. Oracle vous demandera de vous authentifier (identifiant + mot de passe). Si ce n’est pas déjà fait, vous pouvez créer un compte sur Oracle.

Une fois l’installateur téléchargé, double-cliquez sur ce programme. Passez les étapes d’installation en laissant les valeurs par défaut.

Il faut ensuite intégrer **SceneBuilder** à Eclipse.

Pour ce faire, dans Eclipse : Window -> Preferences -> JavaFX



Cliquer sur « **Browse...** » pour ouvrir l’explorateur de fichiers et trouver l’application **SceneBuilder** et renseigner le chemin. Cliquer ensuite sur « **Apply and Close** ».

Pour vérifier que **SceneBuilder** est bien intégré à Eclipse, vous pouvez suivre la démarche suivante :

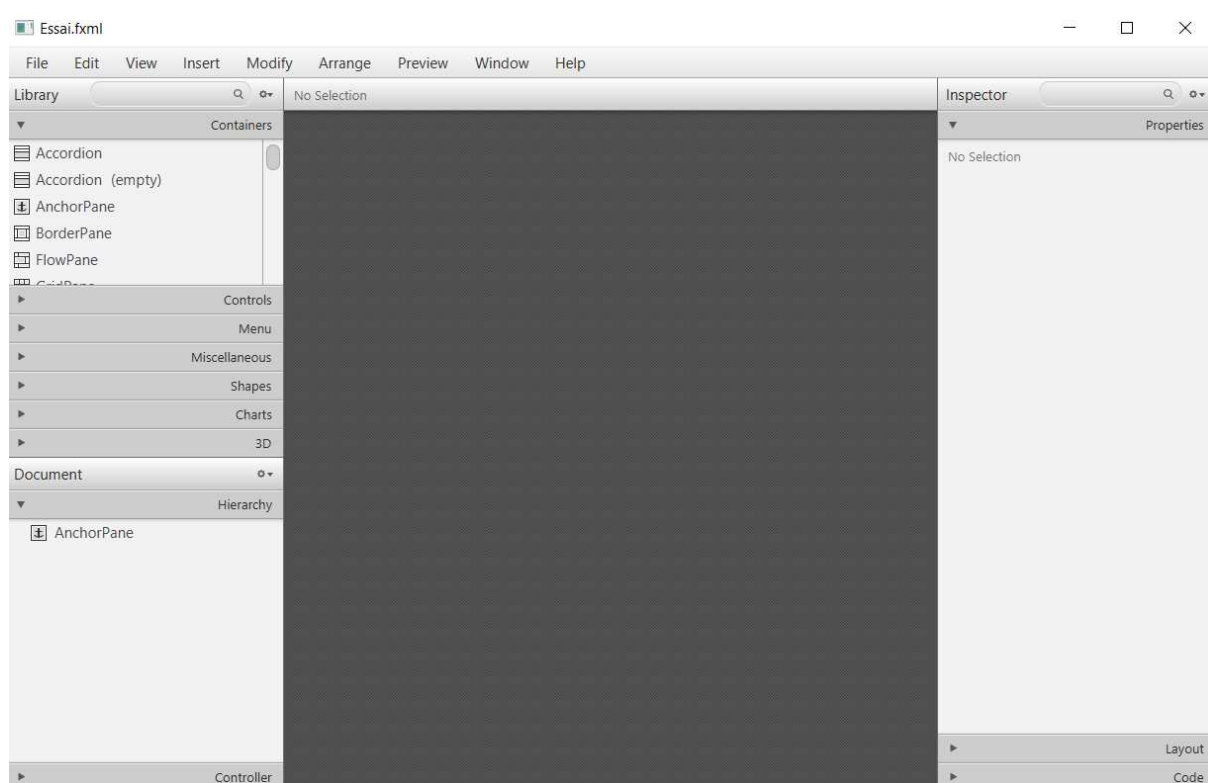
1) créer un fichier **fxml** de la manière suivante :

dans le **Package Explorer**, faire un clic droit sur le package de l’application JavaFX créée précédemment. Puis :

New -> Other... -> JavaFX -> New FXML Document

Cliquer sur “**Next**”. Renseigner le nom d’un fichier, *Essai* par exemple. Puis cliquer sur « **Finish.** ».

2) dans le **Package Explorer**, faire un clic droit sur le fichier **fxml** qui vient d’être ajouté et sélectionner **Open with SceneBuilder**. Vous devez obtenir une fenêtre telle que celle-ci :



**Félicitations !**

**Vous avez réussi à installer votre environnement de travail JavaFX !**

# Introduction à JavaFX

On distingue 2 catégories de programmes selon leur interface avec l'utilisateur :

- Programmes à interface console : le dialogue, entre le programme et l'utilisateur, se fait de manière séquentielle, dans une seule fenêtre, appelée fenêtre console. C'est le programme qui sollicite l'utilisateur au moment voulu, c'est-à-dire le moment où il doit fournir une information.
- Programmes à interface graphique : c'est l'utilisateur qui a l'impression de piloter le programme qui réagit à ses demandes :
  - ✓ Clic sur un bouton
  - ✓ Saisie dans des boîtes de dialogue
  - ✓ Choix dans un menu
  - ✓ Fermeture d'une fenêtre

On parle de programmation événementielle, car le programme réagit à des événements provoqués par l'utilisateur. Le langage Java intègre des outils, sous la forme de classes standard, pour gérer, programmer des interfaces graphiques.

## 1) Introduction

Au début du langage Java, les interfaces graphiques étaient créées en utilisant la bibliothèque AWT (Abstract Window Toolkit). Cette API permettait au programmeur de coder l'interface graphique indépendamment du système d'exploitation. Chaque composant graphique était dessiné et contrôlé par un composant natif spécifique au système d'exploitation (on parle alors de composants lourds).

Des améliorations ont ensuite été apportées avec la librairie Swing qui est apparue pour compléter et remplacer en partie AWT. Avec Swing, les composants sont dits légers, car ils ne font pas appel à des composants natifs. Mais à partir de 2014, le développement de Swing a été arrêté au profit de JavaFX.

JavaFX est une plateforme et une bibliothèque qui permet de développer des interfaces graphiques pour des programmes Java. Ces programmes peuvent être des applications de bureau, des sites internet, des applications mobiles ...

A l'origine c'est la société Sun Microsystems qui a créé JavaFX, en 2008. Puis Oracle l'a racheté et a continué à développer JavaFX jusqu'à la version 11 du JDK. Depuis c'est la communauté OpenJFX qui continue à faire évoluer JavaFX.

La plateforme JavaFX permet d'utiliser 2 techniques complémentaires pour créer les interfaces graphiques des applications : la **manière déclarative** et la **manière procédurale**.



### Avec la manière déclarative :

- l'aspect statique (donc le visuel) de l'interface est décrit dans un fichier **FXML** (la syntaxe repose sur du XML)
- on peut utiliser l'utilitaire **SceneBuilder** pour créer l'interface. Il s'agit d'un outil interactif de *design* capable de produire des fichiers FXML. SceneBuilder a été d'abord développé par Oracle. Depuis 2013, ce logiciel est Open Source et évolue dans le cadre du projet OpenJFX.
- on obtient du code qui obligatoirement sépare la vue des autres traitements (en accord avec le modèle MVC = Modèle – Vue - Contrôleur)
- la gestion des événements, comme le clic sur un bouton par exemple, ou bien l'évolution dynamique de l'interface doit être gérée dans le code Java

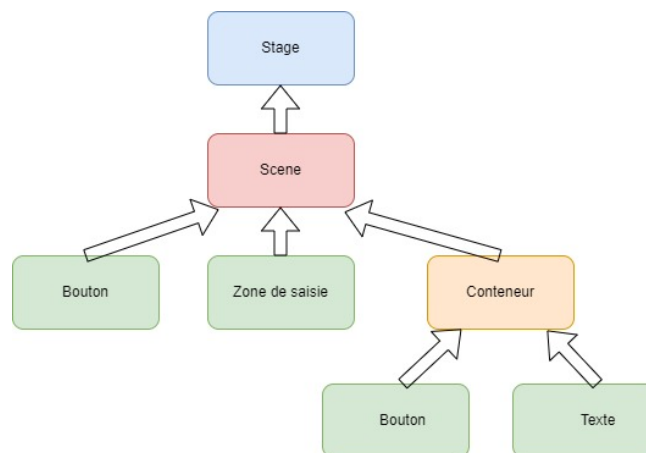
### Avec la manière procédurale :

- on utilise des bibliothèques de **JavaFX** pour coder entièrement l'aspect visuel de l'interface dans le code Java
- comme dans la manière déclarative, la gestion des événements, donc des interactions avec l'utilisateur, ou bien l'évolution dynamique de l'interface doit être gérée dans le code Java

## 2) Concepts de base - Afficher une fenêtre vide

Une application JavaFX est une classe qui doit obligatoirement hériter de la classe prédéfinie **Application** (définie dans le package `javafx.application`).

La fenêtre principale d'une application est représentée par un objet de type **Stage**. On doit ensuite placer dans l'objet **Stage**, une instance de la classe **Scene**. Cet objet de type **Scene** contient l'interface proprement dite. Elle est constituée des différents éléments de l'interface graphique, comme des boutons, des zones de texte, ou des conteneurs qui contiennent eux-mêmes d'autres éléments (des boutons, des zones de saisie, des cases à cocher ...).



**Stage** est une classe JavaFX qui correspond à une fenêtre graphique. Par défaut elle est dotée d'une zone pour afficher un titre (le titre de la fenêtre), de 2 icônes de redimensionnement et d'une croix de fermeture. Nous allons dans un premier temps, examiner un programme qui permet d'afficher une fenêtre comme celle-ci, donc une fenêtre très simple :



Pour ce faire, nous devons coder une classe qui hérite de la classe prédéfinie *Application*. Dans la bibliothèque JavaFX, *Application* est une classe abstraite. Le point d'entrée d'une application JavaFX est une instance de la classe *Application*.

Dans la fonction *main* de la classe ci-dessous, nous trouvons un appel à la méthode *launch*. Cette méthode est définie dans la classe *Application*. Voici les étapes qu'elle va réaliser :

1. créer une instance de la classe qui hérite de *Application*, donc ici une instance de *ExempleCours1*
2. appeler la méthode *init* définie dans cette classe, si elle est présente
3. appeler la méthode *start* de cette classe, et lui passer en paramètre une fenêtre par défaut, celle qui vient d'être créée. Cette fenêtre est de type *Stage*. Ce sera la fenêtre principale de l'application.
4. Attendre que l'application se termine : soit lorsque toutes les fenêtres de l'application auront été fermées, soit si l'application appelle *Platform.exit()*, ce qui est déconseillé
5. Appeler la méthode *stop* de la classe (ici la classe *ExempleCours1*) si elle est présente

```
/*
 * Affiche la fenêtre par défaut de JavaFX
 * ExempleCours1.java
 */
package application;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;

/**
 * Cette classe est la classe principale d'une application JavaFX.
 * On dit aussi le point d'entrée de l'application, car c'est la première classe qui
 * sera exécutée au lancement de l'application (car elle hérite de la classe Application).
 * La fenêtre affichée aura pour titre "Première fenêtre JavaFX", une hauteur de 300 pixels
 * et une largeur de 500 pixels
 * @author C. Servièrès
 */
public class ExempleCours1 extends Application {

    @Override
    public void start(Stage primaryStage) {

        // on définit le titre, la hauteur et la largeur de la fenêtre
        primaryStage.setTitle("Premiere fenetre JavaFX");
        primaryStage.setHeight(300); // l'unité est le pixel
        primaryStage.setWidth(500);

        // on demande à ce que la fenêtre soit affichée
        primaryStage.show();
    }
}
```

```

/**
 * Programme principal
 * @param args argument non utilisé
 */
public static void main(String[] args) {

    /* la méthode launch va créer une instance de la classe courante,
     * donc ExempleCours1
     * Ensuite la méthode start de cette classe est automatiquement appelée
     */
    Launch(args);
}
}

```

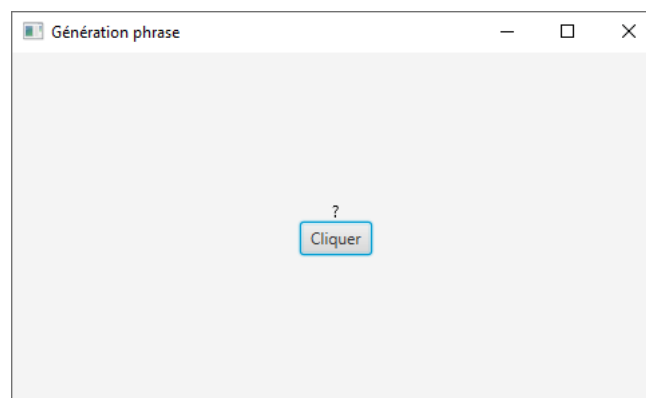
### Exemple 1 – Fenêtre basique – On n'utilise pas de Scene

On constate que dans le code de la méthode *start*, on a renseigné un titre, une hauteur et une largeur pour la fenêtre. On note également qu'il faut demander explicitement son affichage (appel à la méthode *show*).

C'est dans la méthode *start* que l'on doit agir sur le *Stage* et créer la scène (pas dans la méthode *init*)

## 3) Afficher une fenêtre contenant une Scène

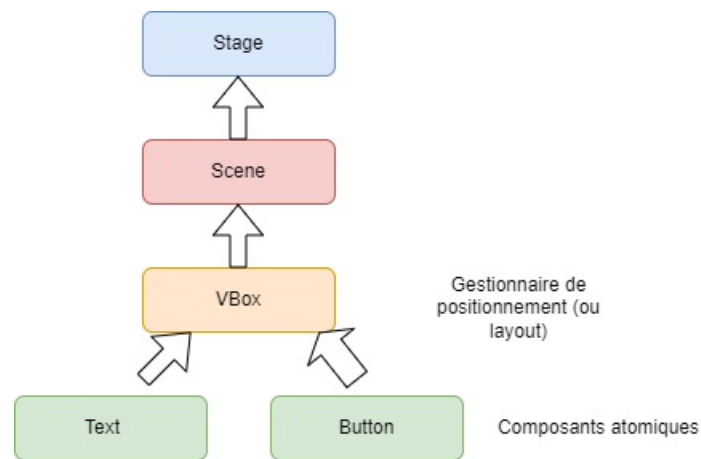
Nous souhaitons maintenant afficher une fenêtre contenant une zone de texte (le caractère ' ? ' dans l'exemple ci-dessous) et d'un bouton ayant le libellé « Cliquer ».



Pour ce faire, nous devons créer une Scène que nous placerons dans le Stage (la fenêtre de base créée par défaut).

Le composant qui permet d'afficher un bouton se nomme **Button**, celui qui permet d'afficher du texte se nomme **Text**. Ces composants ne doivent pas être placés directement dans la **Scène**. Ils doivent d'abord être positionnés dans un gestionnaire de mise en forme qui lui-même sera placé dans la Scène.

Un gestionnaire de mise en forme est un conteneur (dans le sens où il peut contenir d'autres composants). Il n'est pas directement visible. Il sert à définir une politique pour positionner les composants qu'il contient.



Il existe différents types de gestionnaires de positionnement. Ils seront étudiés en détail dans un autre chapitre. Pour cet exemple, nous utilisons *VBox*, le *layout* qui affiche les composants qu'il contient les uns en dessous des autres à la verticale (V = vertical). De plus, nous souhaitons que les composants soient centrés :

```
VBox racine = new VBox();
racine.setAlignment(Pos.CENTER);
```

Les composants atomiques, zone de texte et bouton, sont créés de la manière suivante :

```
Text message = new Text("?");
Button bouton = new Button("Cliquer");
```

Ils sont ensuite ajoutés au *layout*, ou plus exactement ajoutés en tant qu'enfant du *layout*, grâce à un appel à la méthode *add* :

```
racine.getChildren().add(message);
racine.getChildren().add(bouton);
```

Il ne reste plus qu'à créer une *Scène* à laquelle nous associons le *layout* *racine* :

```
Scene scene = new Scene(racine);
primaryStage.setScene(scene);
```

Pour l'instant, le clic sur le bouton n'est pas géré. Si l'on clique, rien ne se passe.

```

/*
 * Affiche une fenêtre JavaFX
 * dotée d'un bouton et d'une zone de texte
 * ExempleCours2.java
 */
package application;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.geometry.Pos;

/**
 * Cette classe est la classe principale d'une application JavaFX.
 * On dit aussi le point d'entrée de l'application, car c'est la première classe qui
 */

```

```

* sera exécutée au lancement de l'application (car elle hérite de la classe Application).
* La fenêtre affichée aura pour titre "Génération phrase", une hauteur de 300 pixels
* et une largeur de 500 pixels.
* La fenêtre est dotée d'un bouton. Un clic sur le bouton provoque l'affichage d'une
* phrase
* @author C. Servières
*
*/
public class ExempleCours2 extends Application {

    /** Phrases successivement affichées dans la zone de texte */
    private static final String[] MESSAGES = {
        "?",
        "Bonjour !",
        "Bienvenue dans cette fenêtre JavaFX !",
        "La présentation est gérée par un layout VBox",
        "Le composant qui affiche ce texte est un Text",
        "Le bouton ci-dessous est de type Button",
        "Au revoir !"};

    /**
     * Index du message courant, celui affiché par la zone de texte
     */
    private int indexMessage;

    @Override
    public void start(Stage primaryStage) {
        indexMessage = 0;

        // on définit le titre, la hauteur et la largeur de la fenêtre
        primaryStage.setTitle("Génération phrase");
        primaryStage.setHeight(300);
        primaryStage.setWidth(500);

        /**
         * La mise en forme de la fenêtre (ou dit autrement la présentation du bouton
         * et de la phrase affichée) sera gérée par un layout de type VBox (V = vertical).
         * Donc les 2 composants, Button et Text, seront affichés verticalement, l'un en
         * dessous de l'autre
         */
        VBox racine = new VBox();

        // les composants seront centrés
        racine.setAlignment(Pos.CENTER);

        // création de la zone de texte
        Text message = new Text("?");

        // création du bouton
        Button bouton = new Button("Cliquer");

        // on ajoute les composants, Text et Button, au layout racine
        racine.getChildren().add(message);
        racine.getChildren().add(bouton);

        // on crée une nouvelle scène
        Scene scene = new Scene(racine);
        primaryStage.setScene(scene);

        // on demande à ce que la fenêtre soit affichée
        primaryStage.show();

    }

    /**
     * Programme principal
     * @param args argument non utilisé
     */
}

```

```

public static void main(String[] args) {

    /* la méthode launch va créer une instance de la classe courante,
     * donc ExempleCours2
     * Ensuite la méthode start de cette classe est automatiquement appelée
     */
    Launch(args);
}
}

```

En résumé, une interface graphique comporte

- des éléments graphiques comme des boutons, des listes, des cases à cocher ... qui permettent de dialoguer avec l'application : saisie d'information, navigation dans le logiciel, visualisation de données .... Ces éléments graphiques sont appelés des **composants atomiques** (ou *contrôles* ou encore *widjets*) dans la terminologie Java.
- des éléments invisibles, par exemple ceux qui servent à agencer les composants les uns par rapport aux autres (ou layouts)
- des fenêtres

## 4) Gérer le clic sur un bouton

La programmation des applications avec interface graphique est basée sur un concept nommé : programmation événementielle. Dans ce type de programmation, les événements déclenchés par l'utilisateur transfèrent le contrôle à des portions de code que l'on nomme « écouteurs d'événement ».

Un événement est une notification que l'interface envoie au programme pour signaler qu'une action s'est produite. Les événements peuvent être par exemple : un clic sur bouton, le fait de cocher une case, la sélection d'un élément dans une liste déroulante...

En JavaFX, les événements sont représentés par des objets de la classe *Event*, ou par l'une des ses sous-classes. De nombreux événements sont prédéfinis sous la forme de sous-classes de *Event*. Chaque événement appartient à une certaine catégorie, nommée *EventType*. Par exemple, si l'utilisateur clique sur un bouton, un événement de la catégorie *Action* va se produire et se propager jusqu'à atteindre une portion de code correspondant à son écouteur.

```

// création du bouton
Button bouton = new Button("Cliquer");

/*
 * On associe au bouton un écouteur d'événement (avec la méthode setOnAction)
 * L'événement écouté est le clic sur le bouton.
 * L'écouteur est une instance d'une classe qui doit implémenter l'interface
 * EventHandler. Cette interface contient la méthode handle. C'est elle qui
 * est appelée automatiquement lors du clic sur le bouton. On code dans cette
 * méthode les opérations à effectuer lors du clic sur le bouton.
 */
bouton.setOnAction((EventHandler<ActionEvent>) new EventHandler<ActionEvent>() {

```

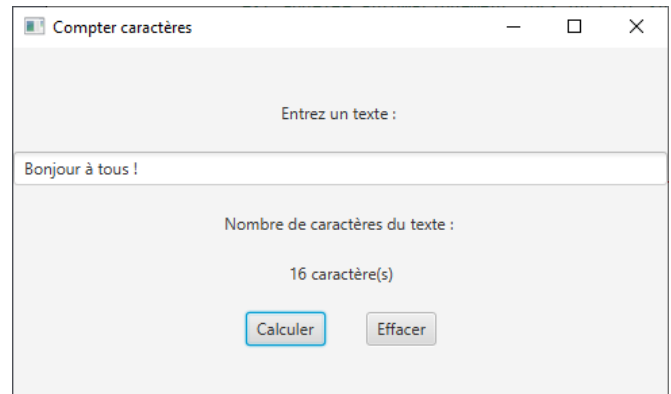
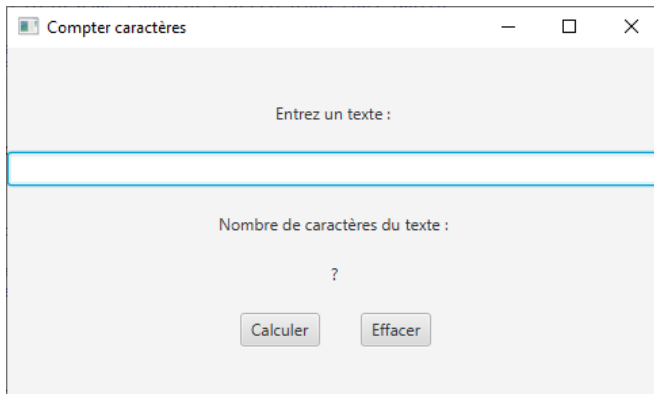
```

@Override
public void handle(ActionEvent event) {

    // on affiche dans le texte, le message suivant
    indexMessage = (indexMessage + 1) % MESSAGES.length;
    message.setText(MESSAGES[indexMessage]);
}
});

```

## 5) Autre exemple



```

/*
 * Affiche une fenêtre JavaFX. Permettra de compter les caractères d'une chaîne
 * ExempleCours3.java
 */

package application;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;

/**
 * Cette classe est la classe principale d'une application JavaFX.
 * On dit aussi le point d'entrée de l'application, car c'est la première classe qui
 * sera exécutée au lancement de l'application (car elle hérite de la classe Application).
 * La fenêtre affichée aura pour titre "Compter caractères", une hauteur de 300 pixels
 * et une largeur de 500 pixels.
 * La fenêtre est dotée d'une zone de saisie dans laquelle l'utilisateur sera invité
 * à saisir une phrase. L'application affichera ensuite, suite au clic sur le bouton
 * "Calculer", le nombre de caractères de cette phrase.
 * Un clic sur le bouton "Effacer" effacera les valeurs affichée et saisie.
 * @author C. Servières
 */

```

```

public class ExempleCours3 extends Application {

    @Override
    public void start(Stage primaryStage) {

        // on définit le titre, la hauteur et la largeur de la fenêtre
        primaryStage.setTitle("Compter caractères");
        primaryStage.setHeight(300);
        primaryStage.setWidth(500);

        /*
         * La mise en forme de la fenêtre sera gérée par un layout de type VBox
         * (V = vertical).
         * Donc les composants, Text, TextField, Text et la ligne contenant les 2 boutons,
         * seront affichés verticalement, les uns en dessous des autres
         */
        VBox racine = new VBox(20); // 20 pixels entre les éléments du VBox

        // les composants seront centrés
        racine.setAlignment(Pos.CENTER);

        // création zone de texte pour inviter l'utilisateur à saisir
        Label invitation = new Label("Entrez un texte : ");

        // création d'une zone de saisie
        TextField zoneSaisieTexte = new TextField();

        // création zone de texte pour expliquer le résultat
        Label explicationResultat = new Label("Nombre de caractères du texte : ");

        // création zone de texte qui contiendra la valeur du résultat
        Label resultat = new Label("?");

        // les 4 composants sont ajoutés au layout racine
        racine.getChildren().addAll(invitation, zoneSaisieTexte, explicationResultat, resultat);

        /*
         * création d'un layout correspondant à de la ligne qui contiendra
         * les boutons Calculer et Effacer. Ceux-ci sont affichés sur une même ligne.
         * Le layout est HBox pour un alignement horizontal (H = horizontal)
         */
        HBox ligneBouton = new HBox(30); // 30 pixels entre les éléments du HBox
        ligneBouton.setAlignment(Pos.CENTER); // pour centrer la ligne

        // création des 2 boutons et ajout au layout ligneBouton
        Button calculer = new Button("Calculer");
        Button effacer = new Button("Effacer");
        ligneBouton.getChildren().addAll(calculer, effacer);

        // le layout ligneBouton est ajouté au layout racine
        racine.getChildren().add(ligneBouton);

        /*
         * On associe au bouton calculer un écouteur d'événement (clic sur bouton)
         * Lors du clic sur le bouton "calculer", on affiche dans le Text resultat
         * le nombre de caractères de la chaîne saisie
         */
        calculer.setOnAction((EventHandler<ActionEvent>) new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {

                // la méthode getText permet de récupérer le texte présent dans zoneSaisieTexte
                resultat.setText(zoneSaisieTexte.getText().length() + " caractère(s)");
            }
        });
    }
}

```



```

    /**
     * On associe au bouton effacer un écouteur d'événement (clic sur bouton)
     * Lors du clic sur le bouton "effacer", on efface le texte saisi par l'utilisateur
     * et on dans le texte de résultat, on place le caractère '?'
     */
    effacer.setOnAction((EventHandler<ActionEvent>) new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            zoneSaisieTexte.setText("");
            resultat.setText("?");
        }
    });

    // on créé une nouvelle scène
    Scene scene = new Scene(racine);
    primaryStage.setScene(scene);

    // on demande à ce que la fenêtre soit affichée
    primaryStage.show();
}

/**
 * Programme principal
 * @param args argument non utilisé
 */
public static void main(String[] args) {
    /** la méthode launch va créer une instance de la classe courante,
     * donc ExempleCours2
     * Ensuite la méthode start de cette classe est automatiquement appelée
     */
    launch(args);
}
}

```

# JavaFX – Les contrôles

Une interface graphique comporte :

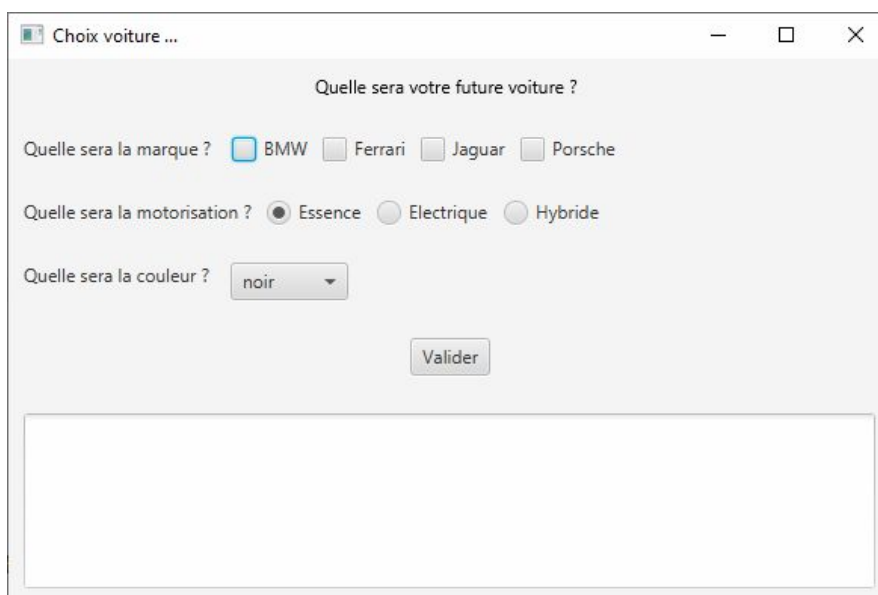
- des éléments graphiques comme des boutons, des listes, des cases à cocher ... qui permettent de dialoguer avec l'application : saisie d'information, navigation dans le logiciel, visualisation de données .... Ces éléments graphiques sont appelés des **composants atomiques** (ou **contrôles** ou encore **widgets**) dans la terminologie Java.
- des éléments invisibles, par exemple ceux qui servent à agencer les composants les uns par rapport aux autres (ou *layouts*)
- des fenêtres

Le but de ce chapitre est de présenter les principaux contrôles utilisables avec JavaFX.

## 1) Les contrôles étiquetés

Un contrôle étiqueté est un contrôle auquel est associé un texte en lecture seule. Les principales classes qui gèrent ces contrôles sont **Label**, **Text**, **Button**, **CheckBox**, **RadioButton** ... Toutes ces classes héritent d'une classe parente nommée **Labeled**.

La fenêtre ci-dessous comporte des **Labels**, des cases à cocher pour choisir la marque de la voiture (plusieurs cases peuvent être cochées), des boutons radios pour choisir la motorisation (un seul choix possible parmi les 3) et un bouton.



The screenshot shows a JavaFX window titled "Choix voiture ...". Inside the window, there is a form titled "Quelle sera votre future voiture ?". The form contains three sections:

- Quelle sera la marque ?**: Four checkboxes for "BMW", "Ferrari", "Jaguar", and "Porsche". The "BMW" checkbox is checked.
- Quelle sera la motorisation ?**: Three radio buttons for "Essence", "Electrique", and "Hybride". The "Essence" radio button is selected.
- Quelle sera la couleur ?**: A dropdown menu showing "noir".

At the bottom of the form is a "Valider" button. Below the button is a large empty rectangular area.

## Label

Un label est une étiquette contenant du texte (ou éventuellement une icône) et qui est utilisée pour décrire un autre composant. Par exemple, on trouve généralement un label à gauche d'un *TextField*, donc une zone de saisie.

Constructeurs	
<i>Label()</i>	
<i>Label(String text)</i>	création du <i>Label</i> avec un texte initial

Méthodes usuelles	
<i>String getText()</i>	Renvoie le texte du label
<i>void setText(String text)</i>	Modifie le texte affiché par le label

## Text

Un composant **Text** ressemble à un composant *Label*. Toutefois, alors que la classe **Label** hérite de la classe *Control* ce qui fait d'un *Label* un contrôle à part entière, la classe **Text** hérite de *Shape*. Un **Text** est donc plus proche d'une forme géométrique. Concrètement, le programmeur aura plus de possibilité pour mettre en forme l'affichage d'un composant **Text** : plus de couleurs, plus de rendus possibles avec des ombres par exemple...

Les constructeurs et les méthodes usuelles d'un composant **Text** sont identiques à ceux d'un **Label**.

## Button

Les constructeurs et les méthodes usuelles d'un composant **Button** sont identiques à ceux d'un **Label**. Dans une autre section, nous étudierons les principales manières de gérer le clic sur un bouton.

## CheckBox

Une case à cocher possède 2 états : cochée ou non cochée (en réalité il y a un 3<sup>ème</sup> état : indéfini).

Constructeurs	
<i>CheckBox()</i>	
<i>CheckBox (String text)</i>	création d'une case à cocher associée au texte argument

Méthodes usuelles	
<i>String getText()</i>	Renvoie le texte de la case à cocher (son libellé)
<i>void setText(String text)</i>	Modifie le texte affiché par la case à cocher

<i>boolean isSelected()</i>	Renvoie vrai si la case est cochée (faux dans le cas contraire)
<i>void setSelected(boolean value)</i>	Fait en sorte que la case soit cochée si le paramètre est égal à vrai (et décochée dans le cas contraire)

### Exemple de création de 4 cases à cocher :

```
// création des cases à cocher
CheckBox caseBMW = new CheckBox("BMW");
CheckBox caseFerrari = new CheckBox("Ferrari");
CheckBox caseJaguar = new CheckBox("Jaguar");
CheckBox casePorsche = new CheckBox("Porsche");
```

## RadioButton

Un bouton radio ressemble à une case à cocher dans le sens où il possède aussi 2 états (sélectionné ou pas). La différence est que les boutons radio fonctionnent en groupe. Le programmeur doit réunir dans un même groupe les boutons radio qui sont liés (qui participent à un même choix, comme par exemple celui de la motorisation de la voiture). Ensuite c'est le groupe qui gère automatiquement le fait qu'à un instant donné un seul bouton radio est sélectionné.

Le programmeur doit donc utiliser aussi la classe **ToggleGroup** qui représente un groupe de boutons radio. Ce composant est invisible. La classe **ToggleGroup** possède un constructeur par défaut. On peut ajouter les boutons radio en faisant appel à la méthode **getToggles** puis à la méthode **add** ou **addAll**, pour ajouter respectivement un bouton radio ou plusieurs boutons radio au groupe.

### Exemple :

```
RadioButton btnEssence = new RadioButton("Essence");
RadioButton btnElectrique = new RadioButton("Electrique");
RadioButton btnHybride = new RadioButton("Hybride");
RadioButton btnEssence.setSelected(true); // sélectionné par défaut

// création d'un groupe de boutons radio
// (l'objectif est qu'un seul puisse être sélectionné)
ToggleGroup groupe = new ToggleGroup();
groupe.getToggles().addAll(btnEssence, btnElectrique, btnHybride);
```

Constructeurs	
<i>RadioButton()</i>	
<i>RadioButton (String text)</i>	création d'un bouton radio associé au texte argument

Méthodes usuelles	
<i>String getText()</i>	Renvoie le texte du bouton radio (son libellé)
<i>void setText(String text)</i>	Modifie le texte affiché par le bouton radio
<i>boolean isSelected()</i>	Renvoie vrai si le bouton radio est sélectionné (faux dans le cas contraire)
<i>void setSelected(boolean value)</i>	Fait en sorte que le bouton radio soit sélectionné si le paramètre est égal à vrai (et décochée dans le cas contraire)
<i>void setToggleGroup(ToggleGroup group)</i>	Ajoute le bouton radio au groupe argument

## 2) Les contrôles pour saisir du texte

L'utilisateur pourra saisir du texte dans un **TextField** si sa saisie doit être faite sur une seule ligne ou dans le cas contraire dans un **TextArea**. Le composant **PasswordField** permet quant à lui de saisir un mot de passe (caractères invisibles). Toutes ces classes héritent de la classe **TextInputControl**.

### **TextField**

Zone de saisie dans laquelle l'utilisateur pourra effectuer une saisie de texte, sur une seule ligne.

Constructeurs	
<i>TextField()</i>	
<i>TextField(String text)</i>	création du <i>TextField</i> avec un texte initial

Méthodes usuelles	
<i>String getText()</i>	Renvoie le texte saisi dans le <i>TextField</i>
<i>void setEditable(boolean value)</i>	Si l'argument est égal à faux, le <i>TextField</i> ne sera pas modifiable par l'utilisateur (car il n'aura pas le focus)
<i>void setMaxWidth(double width)</i>	Pour fixer la largeur maximale du <i>TextField</i>
<i>void setMinWidth(double width)</i>	Pour fixer la largeur minimale du <i>TextField</i>
<i>void setPrefColumnCount(int cols)</i>	Pour fixer la largeur selon le nombre moyen de caractères que pourra contenir le <i>TextField</i>
<i>void setPromptText(String prompt)</i>	Pour fixer le texte d'indication affiché. Celui-ci disparaît dès que l'utilisateur commence une saisie
<i>void setText(String text)</i>	Pour fixer le texte contenu dans le <i>TextField</i>
<i>void appendText(String text)</i>	Concatène le texte argument à la suite du texte contenu dans le <i>TextField</i>

### Remarque :

Si l'utilisateur est supposé saisir un entier dans un **TextField** rien ne garantit qu'il aura bien respecté cette consigne. La méthode **getText** renvoie la valeur saisie sous la forme d'une chaîne de caractères. On peut ensuite la convertir en entier en appelant la méthode **parseInt** de la classe **Integer** :

**Integer.parseInt(la\_chaine)**

Cet appel renvoie un résultat de type *int* si la conversion est possible. Dans le cas contraire, il provoque la levée de l'exception **NumberFormatException**.

Généralement, on aura du code structuré de la manière suivante :

```

int entierSaisi = 0 ;
try {
    entierSaisi = Integer.parseInt(leTextField.getText()) ;
} catch (NumberFormatException erreur) {

    // l'utilisateur a commis une erreur (n'a pas saisi un entier ou
    // n'a rien saisi)
    . . .
}

```

Dans la classe **Double**, il y a une méthode **parseDouble** qui fonctionne sur le même principe.

## TextArea

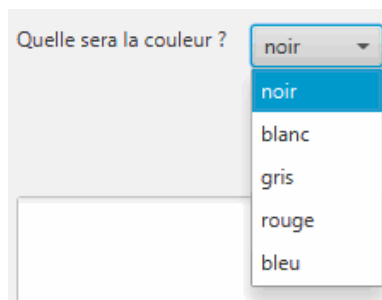
Zone de saisie dans laquelle l'utilisateur pourra effectuer une saisie de texte, sur plusieurs lignes.

Les constructeurs et les méthodes usuelles d'un composant **TextArea** sont identiques à ceux d'un **TextField**.

## 3) Autres composants

### ComboBox

Une boîte combo permet à l'utilisateur de sélectionner un élément dans une liste, comme par exemple une couleur pour la voiture.



#### Constructeurs

*ComboBox<T>()*

Création d'une boîte combo vide et dont les éléments seront de type T (en général String)

#### Méthodes usuelles

*void setEditable(boolean value)*

Si l'argument est égal à vrai, l'utilisateur sera autorisé à saisir son propre choix

<code>void setVisibleRowCount(int value)</code>	Pour fixer le nombre d'éléments affichés (au-delà d'une barre de défilement est automatiquement ajoutée)
<code>void setPromptText(String text)</code>	Pour fixer le texte initialement affiché (indication)
<code>T getValue()</code>	Renvoie la valeur de l'élément sélectionné
<code>void setValue(T value)</code>	Pour fixer l'élément actuellement sélectionné

### Exemple

```
// création de liste des couleurs
ComboBox<String> listeCouleur = new ComboBox<>();
listeCouleur.getItems().addAll("noir", "blanc", "gris", "rouge", "bleu");

// rendre le 1er élément de la liste visible directement
listeCouleur.getSelectionModel().selectFirst();
```

### Autres composants

Nom	Rôle
<i>ListView</i>	Pour présenter une liste dans laquelle l'utilisateur pourra choisir un ou plusieurs éléments
<i>TableView</i>	Gérer un tableau constitué de lignes et colonnes
<i>Menu</i>	Afficher et gérer une barre de menu (en haut de la fenêtre)
<i>HyperLink</i>	Lien hypertexte
<i>ToggleButton</i>	Bouton avec 2 états : sélectionné ou pas

## 4) Hiérarchie d'héritage

Tous les composants graphiques sont liés par une hiérarchie d'héritage dont voici un extrait. Les classes abstraites sont notées en italique.

```

Node
  Parent
    Region
      Control
        Labeled
          TitledPane
            ButtonBase
              MenuButton
              ToggleButton
              RadioButton
              Hyperlink
              CheckBox
              Button
            Label
          TextInputControl
            TextField
            PasswordField
            TextArea

```

## 5) Code de l'exemple

```

/*
 * Exemple d'utilisation des contrôles suivants : CheckBox, RadioButton, ComboBox et TextArea
 * ExempleCours4.java                                01/22
 */
package application;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.control.TextArea;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;

/**
 * Cette classe illustre l'utilisation des contrôles :
 * - CheckBox case à cocher
 * - RadioButton boutons radio
 * - ComboBox boîte combinée
 * - TextArea zone de saisie (ou affichage de texte) comportant plusieurs lignes
 * L'utilisateur sélectionne les caractéristiques de sa future voiture et une synthèse
 * de celles-ci s'affiche dans le TextArea
 *
 * @author C. Servières
 */

```



```

public class ExempleCours4 extends Application {

    /** Case à cocher pour la marque BMW */
    private CheckBox caseBMW;

    /** Case à cocher pour la marque Ferrari */
    private CheckBox caseFerrari;

    /** Case à cocher pour la marque Jaguar*/
    private CheckBox caseJaguar;

    /** Case à cocher pour la marque Porsche */
    private CheckBox casePorsche;

    /** Bouton radio pour la motorisation essence */
    private RadioButton btnEssence;

    /** Bouton radio pour la motorisation électrique */
    private RadioButton btnElectrique;

    /** Bouton radio pour la motorisation hybride */
    private RadioButton btnHybride;

    /** Boîte combo qui affiche les couleurs possibles */
    private ComboBox<String> listeCouleur;

    /** Zone de texte, sur plusieurs lignes qui affichera la phrase de synthèse */
    private TextArea synthese;

    @Override
    public void start(Stage primaryStage) {

        // on définit le titre, la hauteur et la largeur de la fenêtre
        primaryStage.setTitle("Choix voiture ...");
        primaryStage.setHeight(400);
        primaryStage.setWidth(600);

        /*
         * La mise en forme de la fenêtre sera gérée par un layout de type VBox
         * (V = vertical).
         * Donc les composants, Text, TextField, Text et la ligne contenant les 2 boutons,
         * seront affichés verticalement, les uns en dessous des autres
         */
        VBox racine = new VBox(25); // 20 pixels entre les éléments du VBox
        racine.setPadding(new Insets(10, 10, 10, 10));

        // les composants seront centrés
        racine.setAlignment(Pos.CENTER);

        // création zone de texte pour le titre de l'application
        racine.getChildren().add(new Text("Quelle sera votre future voiture ? "));

        // on ajoute la ligne contenant les cases à cocher
        racine.getChildren().add(preparationLigneCaseACocher());

        // on ajoute la ligne contenant les boutons radio pour choisir la motorisation
        racine.getChildren().add(preparationLigneBoutonRadio());

        // on ajoute la ligne contenant la ComboBox
        racine.getChildren().add(preparationLigneComboBox());

        // création du bouton et ajout à la racine
        Button boutonValider = new Button("Valider");
        boutonValider.setOnAction(event->gererClicValider());
        racine.getChildren().add(boutonValider);

        // création du TextArea et ajout à la racine
        synthese = new TextArea();
        synthese.setEditable(false);
    }
}

```

```

        racine.getChildren().add(synthese);

        // on créé une nouvelle scène
        Scene scene = new Scene(racine);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**
     * Préparation de la ligne qui contient les cases à cocher (et le libellé)
     * @return un composant HBox contenant le libellé et les cases à cocher
     */
    private HBox preparationLigneCaseACocher() {

        // HBox qui sera renvoyé
        HBox ligne = new HBox(10);
        ligne.getChildren().add(new Label("Quelle sera la marque ? "));

        // création des cases à cocher
        caseBMW = new CheckBox("BMW");
        caseFerrari = new CheckBox("Ferrari");
        caseJaguar = new CheckBox("Jaguar");
        casePorsche = new CheckBox("Porsche");

        // ajout des cases à la ligne
        ligne.getChildren().addAll(caseBMW, caseFerrari, caseJaguar, casePorsche);

        return ligne;
    }

    /**
     * Préparation de la ligne qui contient les boutons radio (et le libellé)
     * pour choisir la motorisation
     * @return un composant HBox contenant le libellé et les boutons radio
     */
    private HBox preparationLigneBoutonRadio() {

        // HBox qui sera renvoyé
        HBox ligne = new HBox(10);
        ligne.getChildren().add(new Label("Quelle sera la motorisation ?"));

        // création des boutons radio
        btnEssence = new RadioButton("Essence");
        btnElectrique = new RadioButton("Electrique");
        btnHybride = new RadioButton("Hybride");
        btnEssence.setSelected(true); // sélectionné par défaut

        // création d'un groupe de boutons radio
        // (l'objectif est qu'un seul puisse être sélectionné)
        ToggleGroup groupe = new ToggleGroup();
        groupe.getToggles().addAll(btnEssence, btnElectrique, btnHybride);

        // ajout des boutons radio à la ligne
        ligne.getChildren().addAll(btnEssence, btnElectrique, btnHybride);

        return ligne;
    }

    /**
     * Préparation de la ligne qui contient la ComboBox pour choisir la couleur
     * de la voiture
     * @return un composant HBox contenant le libellé et la ComboBox
     */
    private HBox preparationLigneComboBox() {

        // HBox qui sera renvoyé
        HBox ligne = new HBox(10);
        ligne.getChildren().add(new Label("Quelle sera la couleur ? "));

        // création de liste des couleurs
    }

```

```

listeCouleur = new ComboBox<>();
listeCouleur.getItems().addAll("noir", "blanc", "gris", "rouge", "bleu");

// rendre le 1er élément de la liste visible directement
listeCouleur.getSelectionModel().selectFirst();

// ajout de la ComboBox à la ligne
ligne.getChildren().add(listeCouleur);

return ligne;
}

/**
 * Gestion du clic sur le bouton Valider. Les choix de l'utilisateur sont
 * récupérés pour former une chaîne de caractères qui est ensuite affichée
 * dans le widget de type TextArea
 */
public void gererClicValider() {
    // aAfficher est la chaîne qui sera affichée dans le TextArea
    String aAfficher = "Vous souhaitez une voiture de marque ";

    // on concatène les marques de voitures associées aux cases cochées
    if (caseBMW.isSelected()) {
        aAfficher += "BMW, ";
    }
    if (caseFerrari.isSelected()) {
        aAfficher += "Ferrari, ";
    }
    if (caseJaguar.isSelected()) {
        aAfficher += "Jaguar, ";
    }
    if (casePorsche.isSelected()) {
        aAfficher += "Porsche, ";
    }

    aAfficher += "\n\navec une motorisation ";

    /**
     * Un seul bouton radio précisant la motorisation doit être sélectionné.
     * On récupère son libellé pour le concaténer à la chaîne
     */
    if (btnEssence.isSelected()) {
        aAfficher += "essence ";
    } else if (btnElectrique.isSelected()) {
        aAfficher += "électrique ";
    } else {
        aAfficher += "hybride ";
    }

    // on récupère la couleur sélectionnée dans la ComboBox
    aAfficher += "\n\net sa couleur sera " + listeCouleur.getValue();

    // le texte constitué est affiché dans le TextArea, synthèse
    synthese.setText(aAfficher);
}

/**
 * Programme principal : lancement de l'application
 * @param args argument non utilisé
 */
public static void main(String[] args) {
    Launch(args);
}
}

```

## 6) Associer un écouteur à un contrôle

Nous avons vu dans le chapitre précédent comment associer un écouteur à un bouton, dans le but de traiter les clics effectués par l'utilisateur.

De la même manière, nous pouvons associer un écouteur aux autres contrôles, pour savoir par exemple quelle **CheckBox** a été cochée, quel bouton radio a été sélectionné ou quelle valeur a été choisie dans une liste combo.

La méthode permettant d'associer un écouteur est **setOnAction** :

**setOnAction(Event Handler<ActionEvent> handler)**

**Ecouteur sur les cases à cocher :**

```
// on associe un écouteur pour prendre en compte le fait qu'une case est cochée
caseBMW.setOnAction(e -> {texteMarque += "BMW, "; });
caseFerrari.setOnAction(e -> {texteMarque += "Ferrari, "; });
caseJaguar.setOnAction(e -> {texteMarque += "Jaguar, "; });
casePorsche.setOnAction(e -> {texteMarque += "Porsche, "; });
```

**Ecouteur sur les boutons radio :**

```
// on associe un écouteur pour prendre en compte le fait qu'un
// bouton radio est sélectionné
btnEssence.setOnAction(e -> {texteMotorisation = "Essence"; });
btnElectrique.setOnAction(e -> {texteMotorisation = "Electrique"; });
btnHybride.setOnAction(e -> {texteMotorisation = "Hybride"; });
```

**Ecouteur sur la liste combo :**

```
// gestion du choix de l'utilisateur
listeCouleur.setOnAction(e -> {texteCouleur = listeCouleur.getValue(); });
```

En utilisant ce principe, l'exemple précédent pourrait être codé de la manière suivante. L'idée est de construire la chaîne qui sera affichée lors du clic sur le bouton « Valider », au fur et à mesure des changements opérés par l'utilisateur. Lorsqu'il coche une case, sélectionne un bouton radio, ou choisit une couleur dans la liste, on concatène à la chaîne le choix qu'il a effectué.

```
/*
 * Exemple d'utilisation des contrôles suivants :
 *         CheckBox, RadioButton, ComboBox et TextArea
 * Des écouteurs sont associés à ces contrôles
 * ExempleCours6.java
 */
package application;

import . . .

/**
 * Cette classe illustre l'utilisation des contrôles :
 * - CheckBox case à cocher
 * - RadioButton boutons radio
 * - ComboxBox boîte combinée
 */
```

```

*   - TextArea      zone de saisie (ou affichage de texte) comportant plusieurs lignes
*   L'utilisateur sélectionne les caractéristiques de sa future voiture et une synthèse
*   de celles-ci s'affiche dans le TextArea
*
* Principe : des écouteurs sont associés aux cases à cocher, radios bouton
*            et à la liste Combo
*            afin d'être informé des changements apportés par l'utilisateur
* @author C. Servières
*
*/
public class ExempleCours6 extends Application {

    . . . . . idem . . . . .

    /** Marque de la voiture sélectionnée */
    private String texteMarque;

    /** Motorisation de la voiture sélectionnée */
    private String texteMotorisation;

    /** Couleur de la voiture sélectionnée */
    private String texteCouleur;

    @Override
    public void start(Stage primaryStage) {

        . . . . . idem . . . . .

        texteMarque = "";
        texteMotorisation = "Essence";
        texteCouleur = "";
    }

    /**
     * Préparation de la ligne qui contient les cases à cocher (et le libellé)
     * @return un composant HBox contenant le libellé et les cases à cocher
     */
    private HBox preparationLigneCaseACocher() {

        // HBox qui sera renvoyé
        HBox ligne = new HBox(10);
        ligne.getChildren().add(new Label("Quelle sera la marque ? "));

        // création des cases à cocher
        caseBMW = new CheckBox("BMW");
        caseFerrari = new CheckBox("Ferrari");
        caseJaguar = new CheckBox("Jaguar");
        casePorsche = new CheckBox("Porsche");

        // ajout des cases à la ligne
        ligne.getChildren().addAll(caseBMW, caseFerrari, caseJaguar, casePorsche);

        // on associe un écouteur pour prendre en compte le fait qu'une case est cochée
        caseBMW.setOnAction(e -> {texteMarque += "BMW, "; });
        caseFerrari.setOnAction(e -> {texteMarque += "Ferrari, "; });
        caseJaguar.setOnAction(e -> {texteMarque += "Jaguar, "; });
        casePorsche.setOnAction(e -> {texteMarque += "Porsche, "; });

        return ligne;
    }
}

```

```

/**
 * Préparation de la ligne qui contient les boutons radio (et le libellé)
 * pour choisir la motorisation
 * @return un composant HBox contenant le libellé et les boutons radio
 */
private HBox preparationLigneBoutonRadio() {

    // HBox qui sera renvoyé
    HBox ligne = new HBox(10);
    ligne.getChildren().add(new Label("Quelle sera la motorisation ?"));

    // création des boutons radio
    btnEssence = new RadioButton("Essence");
    btnElectrique = new RadioButton("Electrique");
    btnHybride = new RadioButton("Hybride");
    btnEssence.setSelected(true); // sélectionné par défaut

    // création d'un groupe de boutons radio
    // (l'objectif est qu'un seul puisse être sélectionné)
    ToggleGroup groupe = new ToggleGroup();
    groupe.getToggles().addAll(btnEssence, btnElectrique, btnHybride);

    // ajout des boutons radio à la ligne
    ligne.getChildren().addAll(btnEssence, btnElectrique, btnHybride);

    // on associe un écouteur pour prendre en compte le fait qu'un
    // bouton radio est sélectionné
    btnEssence.setOnAction(e -> {texteMotorisation = "Essence"; });
    btnElectrique.setOnAction(e -> {texteMotorisation = "Electrique"; });
    btnHybride.setOnAction(e -> {texteMotorisation = "Hybride"; });

    return ligne;
}

/**
 * Préparation de la ligne qui contient la ComboBox pour choisir la couleur
 * de la voiture
 * @return un composant HBox contenant le libellé et la ComboBox
 */
private HBox preparationLigneComboBox() {

    // HBox qui sera renvoyé
    HBox ligne = new HBox(10);
    ligne.getChildren().add(new Label("Quelle sera la couleur ? "));

    // création de liste des couleurs
    listeCouleur = new ComboBox<>();
    listeCouleur.getItems().addAll("noir", "blanc", "gris", "rouge", "bleu");

    // rendre le 1er élément de la liste visible directement
    listeCouleur.getSelectionModel().selectFirst();

    // ajout de la ComboBox à la ligne
    ligne.getChildren().add(listeCouleur);

    // gestion du choix de l'utilisateur
    listeCouleur.setOnAction(e -> {texteCouleur = listeCouleur.getValue(); });

    return ligne;
}

```

```

/**
 * Gestion du clic sur le bouton Valider. Les choix de l'utilisateur sont
 * récupérés pour former une chaîne de caractères qui est ensuite affichée
 * dans le widget de type TextArea
 */
public void gererClicValider() {
    // aAfficher est la chaîne qui sera affichée dans le TextArea
    String aAfficher = "Vous souhaitez une voiture de marque "
        + texteMarque
        + "\n\navec une motorisation "
        + texteMotorisation
        + "\n\net sa couleur sera "
        + texteCouleur;

    // le texte constitué est affiché dans le TextArea, synthèse
    synthese.setText(aAfficher);

    // réinitialisation des choix de l'utilisateur et des widgets
    texteMarque = "";
    texteMotorisation = "Essence";
    texteCouleur = "";
    caseBMW.setSelected(false);
    caseFerrari.setSelected(false);
    caseJaguar.setSelected(false);
    casePorsche.setSelected(false);
    btnEssence.setSelected(true);
    listeCouleur.getSelectionModel().selectFirst();
}

. . . . . idem . . . . .

```

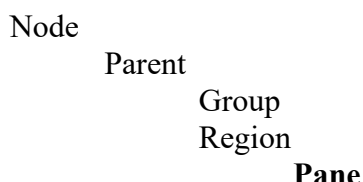
# JavaFX – Les conteneurs layout

## 1) Introduction

D'une manière générale, dans le domaine de la programmation des interfaces graphiques, le rôle d'un *layout* est de gérer le positionnement des composants dans un conteneur. En JavaFX, les *layout-pane* sont des classes spécifiques dont le but est d'appliquer une politique de positionnement précise des nœuds (ou éléments) enfants placés dans le conteneur *layout-pane*.

Ce n'est pas une tâche simple : par exemple, que se passe-t-il si on agrandit une fenêtre ? Les composants doivent-ils s'agrandir ? Si oui, faut-il tous les agrandir ? Si non, faut-il ajouter de l'espace entre eux ? Il est important qu'il y ait des règles bien précises pour gérer la disposition des composants.

Les conteneurs (ou *Layout-Pane*) héritent de la classe *Pane* qui elle-même se situe ainsi sur le graphe d'héritage :



Les principales classes de conteneur sont : *HBox*, *VBox*, *BorderPane*, *StackPane*, *TextFlow*, *AnchorPane*, *TilePane*, *GridPane*, *FlowPane*.

La classe *Region* est la classe parente des composants qu'ils soient des contrôles ou des conteneurs. Elle définit des propriétés en lien avec l'aspect visuel comme les marges, les bordures, le *padding* (avec la notion d'*Insets*). Un objet de type *Insets* permet de définir la valeur du *padding* dans les 4 directions, à l'intérieur du *widget* : haut, droite, bas et gauche, dans cet ordre là). Par exemple :

```
new Insets(15, 15, 15, 15)
```

Dans les exemples de ce chapitre, nous supposons que nous avons créé un tableau contenant des boutons :

```
Button[] lesBoutons;  
lesBoutons = new Button[NB_BOUTON];  
for (int numero = 1 ; numero <= NB_BOUTON; numero++) {  
    lesBoutons[numero - 1] = new Button("Bouton " + numero);  
}
```



Les boutons seront disposés dans la scène en utilisant différents **Layout-Pane** pour la racine :

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

Notons qu'un **layout-pane** peut lui-même contenir un ou plusieurs autres **layout-panes**.

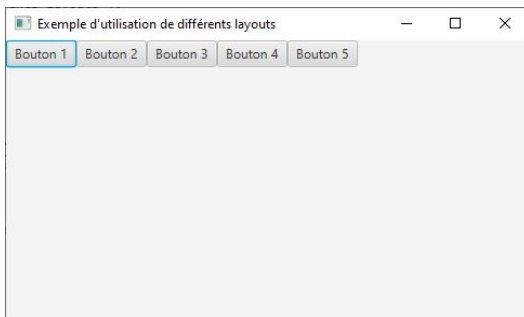
## 2) HBox et VBox

### HBox

Pour commencer, plaçons 5 boutons dans un **HBox** :

```
HBox racine = new HBox();
for (int i = 0; i < 5; i++) {
    racine.getChildren().add(lesBoutons[i]);
}
```

Nous obtenons la disposition suivante dans laquelle les boutons sont placés l'un à la suite de l'autre sur une même ligne :



Si le nombre de boutons est trop grand pour que tous soient affichés sur la ligne, l'affichage sera le suivant :



### VBox

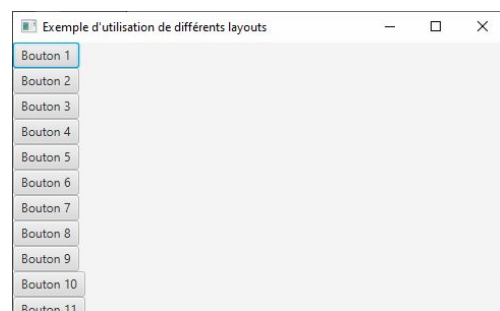
Plaçons 5 boutons dans un **VBox** :

```
VBox racine = new VBox();
for (int i = 0; i < 5; i++) {
    racine.getChildren().add(lesBoutons[i]);
}
```

Nous obtenons la disposition suivante dans laquelle les boutons sont placés l'un en dessous de l'autre sur une même colonne :



Si le nombre de boutons est trop grand pour que tous soient affichés sur la colonne, l'affichage sera le suivant. Seuls les premiers sont visibles. Pour voir les autres, il faut agrandir la fenêtre :



### 3) StackPane

Plaçons 5 boutons dans un **StackPane** :

```
StackPane racine = new StackPane();  
for (int i = 0; i < 5; i++) {  
    racine.getChildren().add(lesBoutons[i]);  
}
```



Les boutons ont été empilés les uns au dessus des autres. Seul le dernier bouton empilé est visible.

### 4) BorderPane

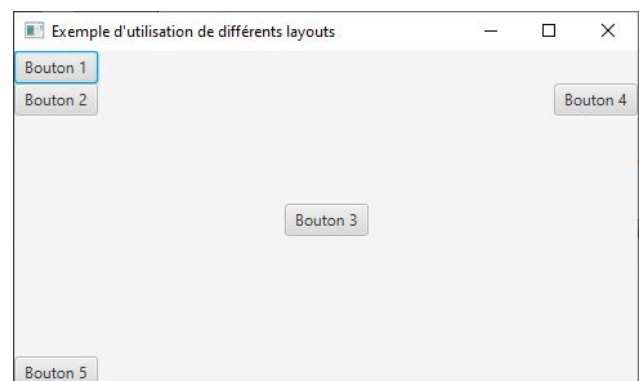
Avec un **BorderPane**, l'espace est découpé en 5 zones : haut, bas, gauche, droite et centre. Le programmeur peut placer un élément dans chacune des 5 zones. Mais certaines zones peuvent rester vides.



Par exemple, on place 5 boutons avec un alignement par défaut, avec le code ci-dessous :

```
BorderPane racine = new BorderPane();  
racine.setTop(lesBoutons[0]);  
racine.setLeft(lesBoutons[1]);  
racine.setCenter(lesBoutons[2]);  
racine.setRight(lesBoutons[3]);  
racine.setBottom(lesBoutons[4]);
```

On obtient :



Par défaut, le bouton placé dans la zone « top » a été aligné avec la constante `Pos.POS_LEFT`, et les autres boutons avec l'alignement suivant :

- zone « bottom » avec alignement `Pos.BOTTOM_LEFT`
- zone « left » avec alignement `Pos.TOP_LEFT`
- zone « right » avec alignement `Pos.TOP_RIGHT`
- zone « center » avec alignement `Pos.CENTER`

Pour mieux comprendre, nous modifions l'alignement par défaut de la manière suivante :

```
BorderPane racine = new BorderPane();

// bouton 1 : alignement au centre de la zone "haut"
racine.setTop(lesBoutons[0]);
BorderPane.setAlignment(lesBoutons[0], Pos.CENTER);

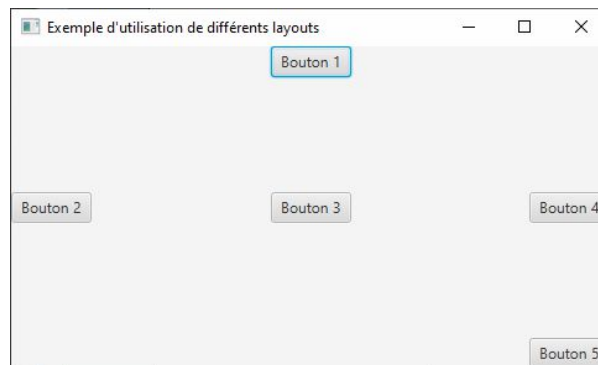
// bouton 2 : alignement au centre de la zone "gauche"
racine.setLeft(lesBoutons[1]);
BorderPane.setAlignment(lesBoutons[1], Pos.CENTER);

// bouton 3 : alignement par défaut dans la zone "centre"
racine.setCenter(lesBoutons[2]);

// bouton 4 : alignement au centre de la zone "droite"
racine.setRight(lesBoutons[3]);
BorderPane.setAlignment(lesBoutons[3], Pos.CENTER);

// bouton 5 : alignement en bas à droite de la zone "bas"
racine.setBottom(lesBoutons[4]);
BorderPane.setAlignment(lesBoutons[4], Pos.BOTTOM_RIGHT);
```

On obtient :



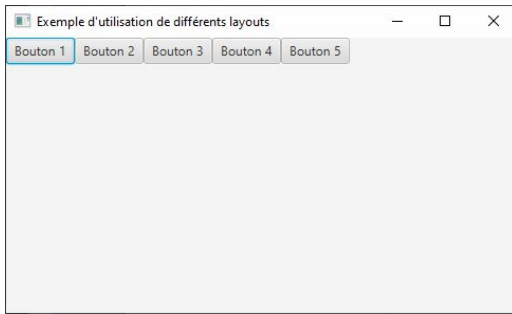
## 5) FlowPane

Avec un **FlowPane**, les composants sont placés les uns à la suite des autres, d'abord sur la ligne courante, puis sur la ligne suivante si l'espace est insuffisant.

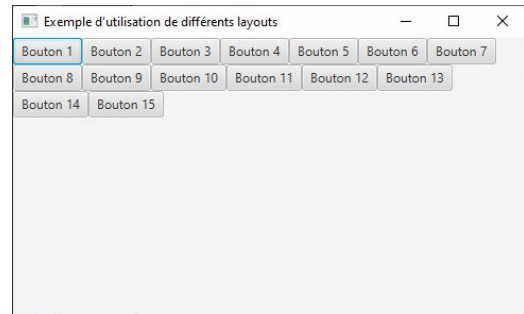
Par exemple, plaçons 5 boutons dans un **FlowPane** :

```
FlowPane racine = new FlowPane();
for (int i = 0; i < 5; i++) {
    racine.getChildren().add(lesBoutons[i]);
}
```

Nous obtenons :



Si nous plaçons 15 boutons, nous obtenons :



## 6) GridPane

Avec un **GridPane**, les composants sont placés en fonction de coordonnées : numéro de ligne et numéro de colonne.

```
GridPane racine = new GridPane();

lesBoutons[0].setText("Premier bouton");
GridPane.setRowIndex(lesBoutons[0], 0);
GridPane.setColumnIndex(lesBoutons[0], 0);

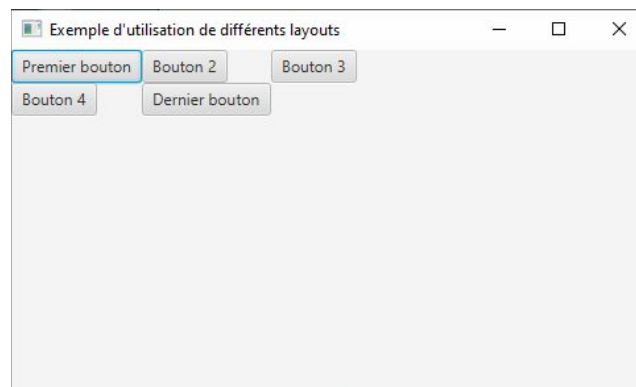
GridPane.setRowIndex(lesBoutons[1], 0);
GridPane.setColumnIndex(lesBoutons[1], 1);

GridPane.setRowIndex(lesBoutons[2], 0);
GridPane.setColumnIndex(lesBoutons[2], 2);

GridPane.setRowIndex(lesBoutons[3], 1);
GridPane.setColumnIndex(lesBoutons[3], 0);

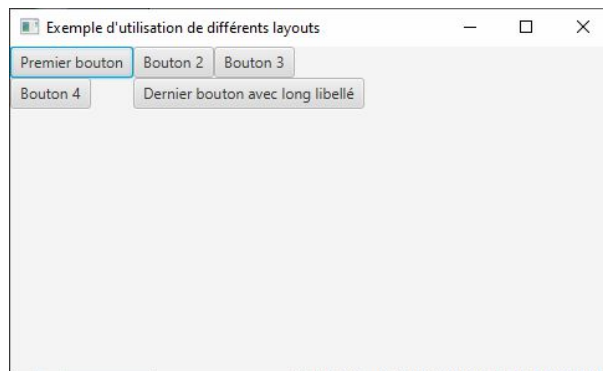
lesBoutons[4].setText("Dernier bouton");
GridPane.setRowIndex(lesBoutons[4], 1);
GridPane.setColumnIndex(lesBoutons[4], 1);

for (int i = 0; i < 5; i++) {
    racine.getChildren().add(lesBoutons[i]);
}
```



Grâce à la propriété *span*, il est possible de dire qu'un bouton doit occuper en largeur plusieurs colonnes (le même principe existe aussi pour les lignes) :

```
lesBoutons[4].setText("Dernier bouton avec long libellé");
GridPane.setRowIndex(lesBoutons[4], 1);
GridPane.setColumnIndex(lesBoutons[4], 1);
GridPane.setColumnSpan(lesBoutons[4], 2);
```



Il est possible d'ajouter un *padding* autour du **GridLayout** (tout comme pour tous les *layouts*), ainsi qu'un espacement horizontal entre les colonnes et un espacement vertical entre les lignes :

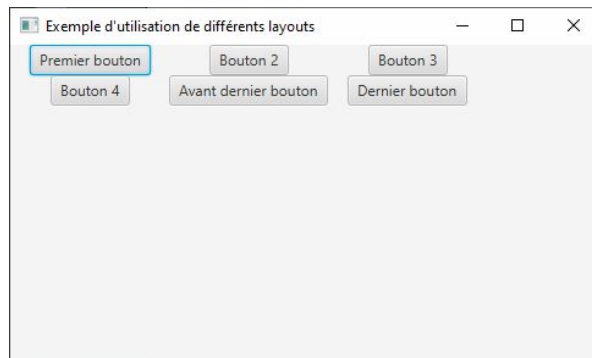
```
racine.setPadding(new Insets(15, 15, 15, 15));
racine.setHgap(25);
racine.setVgap(20);
```

**GridLayout** est souvent utilisé pour des présentations dans lesquelles les contrôles sont disposés en respectant un alignement en colonne. Avec un peu d'entraînement, il a l'avantage d'être simple à utiliser et de conduire à des dispositions rigoureuses.

## 7) *TilePane*

**TilePane** fonctionne de manière très semblable à un **FlowPane**. Chaque élément enfant est placé dans une grille dans laquelle toutes les cellules ont la même taille.

```
TilePane racine = new TilePane();
lesBoutons[0].setText("Premier bouton");
lesBoutons[4].setText("Avant dernier bouton");
lesBoutons[5].setText("Dernier bouton");
for (int i = 0; i < 6; i++) {
    racine.getChildren().add(lesBoutons[i]);
}
```



## 8) *AnchorPane*

**AnchorPane** permet de positionner (ou ancrer) les composants enfants à une certaine distance de ses bordures : haut, bas, droite et gauche. Plusieurs composants peuvent être ancrés à un même bord, et un composant peut être ancré par rapport à plusieurs bords.

Utilisé conjointement avec SceneBuilder, ce **layout** permet de placer rapidement des composants.

# Autres notions

## Boîtes de dialogue et styles

### 1) Boîtes de dialogue

JavaFX propose 3 classes prédéfinies pour gérer des boîtes de dialogues. Elles héritent de la classe parente **Dialog**, elle-même générique et paramétrée par le type T du résultat renvoyé par la boîte de dialogue, lors de sa fermeture.

```
Dialog<T>  
    Alert  
    TextInputDialog  
    ChoiceDialog
```

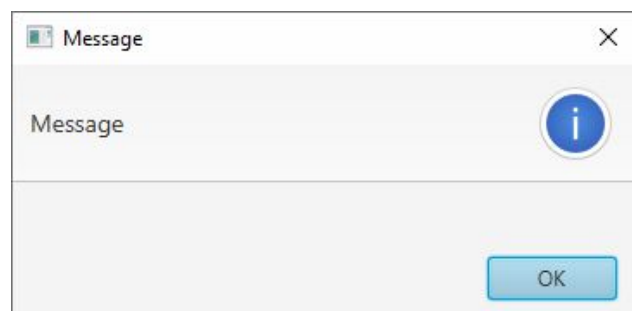
Pour informer l'utilisateur en lui affichant un simple texte, on utilise la classe **Alert**. Pour lui demander de saisir un texte ou de sélectionner un choix parmi une liste d'options, on utilisera respectivement **TextInputDialog** et **ChoiceDialog**.

Notons aussi que le programmeur peut entièrement coder une boîte de dialogue si aucune classe prédéfinie ne lui convient.

#### 1.1) Les boîtes de type **Alert**

Les boîtes d'alerte se subdivisent en plusieurs catégories : Information, Avertissement (warning), Erreur (error)

**Par défaut, une boîte d'alerte se présente de la manière suivante :**



- une zone de titre
- une région d'entête (header), ici avec le mot Message (par défaut) et le logo « i »
- une région de pied de page (footer), ici avec le bouton OK

De plus, entre l'entête et le pied de page, on peut trouver une région de contenu (absente sur l'exemple).

Les 2 lignes de code suivantes permettent d'afficher cette boîte d'information par défaut :

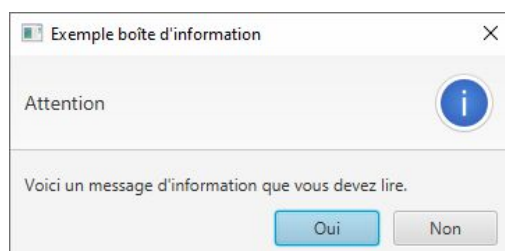
```
Alert boiteAlerte = new Alert(Alert.AlertType.INFORMATION);
boiteAlerte.showAndWait();
```

## Méthodes usuelles

Constructeurs	
<i>Alert(Alert.AlertType a)</i>	Création d'une boîte d'alerte du type spécifié
<i>Alert(Alert.AlertType a, String c, ButtonType ...b)</i>	Création d'une boîte d'alerte du type spécifié, le texte argument est celui affiché en tant que contenu et les derniers arguments indiquent le type du ou des boutons affichés

Méthodes usuelles	
<i>void setAlertType(Alert.AlertType a)</i>	Pour fixer un type d'alerte précis
<i>void setContentText(String a)</i>	Pour fixer le texte affiché en tant que contenu
<i>void setTitle(String text)</i>	Pour fixer le titre
<i>void setHeaderText(String text)</i>	Pour fixer le texte d'entête
<i>void setHeaderText(String text)</i>	Pour fixer l'élément actuellement sélectionné
<i>Optional&lt;R&gt; showAndWait()</i>	Affiche la boîte et attend la réponse de l'utilisateur
<i>void initModality(Modality modalité)</i>	Pour fixer la modalité de la boîte d'alerte. Par défaut, elle est modale (valeurs pour l'argument : NONE, WINDOW_MODAL, APPLICATION_MODAL)

## Autre exemple



```
Alert boiteAlerte =
    new Alert(Alert.AlertType.INFORMATION,
        "Voici un message d'information que vous devez lire.",
        ButtonType.YES, ButtonType.NO);

boiteAlerte.setTitle("Exemple boîte d'information");
boiteAlerte.setHeaderText("Attention");
boiteAlerte.showAndWait();
```

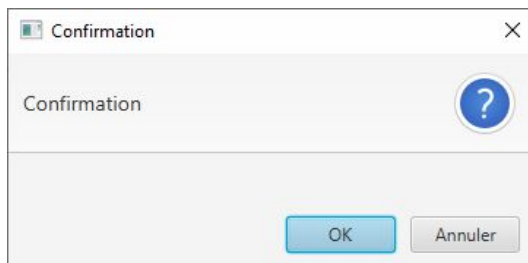
## Autres types de boîte d'alerte



```
Alert boiteAlerte = new  
Alert(Alert.AlertType.WARNING);
```



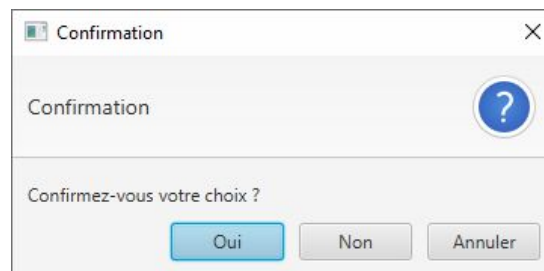
```
Alert boiteAlerte = new  
Alert(Alert.AlertType.ERROR);
```



```
Alert boiteAlerte = new  
Alert(Alert.AlertType.CONFIRMATION);
```

## Comment récupérer le choix de l'utilisateur ?

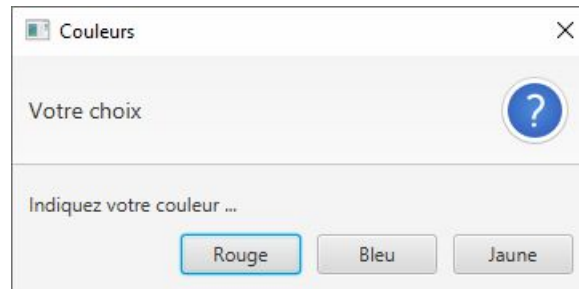
On souhaite savoir sur quel bouton l'utilisateur a cliqué.



```
/*  
 * Création d'une boîte d'alerte de type confirmation.  
 * Elle est dotée de 3 boutons : oui, non, et annuler  
 */  
Alert boiteAlerte = new Alert(Alert.AlertType.CONFIRMATION,  
    "Confirmez-vous votre choix ?",  
    ButtonType.YES, ButtonType.NO, ButtonType.CANCEL);  
  
Optional<ButtonType> option = boiteAlerte.showAndWait();  
if (option.get() == ButtonType.YES) {           // clic sur "oui"  
    realisation.setText("choix confirmé");  
} else if (option.get() == ButtonType.NO) {      // clic sur "non"  
    realisation.setText("choix non confirmé");  
} else if (option.get() == ButtonType.CANCEL) {  
  
    // clic sur "annuler" ou bien sur la croix de fermeture  
    realisation.setText("annulation");  
}
```



## Comment personnaliser les boutons ?



```
/*
 * Création d'une boîte d'alerte de type confirmation.
 * Elle sera dotée de 3 boutons : rouge, bleu et jaune
 */
Alert boiteAlerte = new Alert(Alert.AlertType.CONFIRMATION);
boiteAlerte.setTitle("Couleurs");
boiteAlerte.setHeaderText("Votre choix");
boiteAlerte.setContentText("Indiquez votre couleur ...");

ButtonType rouge = new ButtonType("Rouge");
ButtonType bleu = new ButtonType("Bleu");
ButtonType jaune = new ButtonType("Jaune");

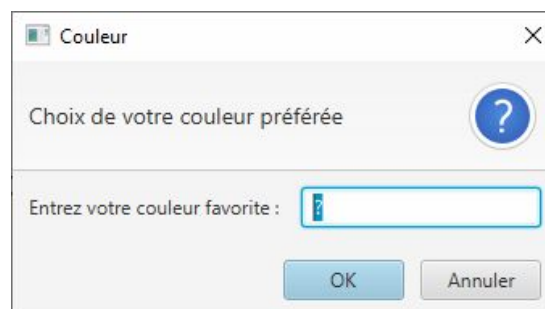
// on supprime les boutons affichés par défaut
boiteAlerte.getButtonTypes().clear();

// on ajoute les 3 boutons personnalisés
boiteAlerte.getButtonTypes().addAll(rouge, bleu, jaune);

Optional<ButtonType> option = boiteAlerte.showAndWait();
if (option.get() == rouge) { // clic sur "rouge"
    realisation.setText("couleur = rouge");
} else if (option.get() == bleu) { // clic sur "bleu"
    realisation.setText("couleur = bleu");
} else if (option.get() == jaune) { // clic sur "jaune"
    realisation.setText("couleur = jaune");
}
```

### 1.2) Les boîtes de type *TextInputDialog*

Automatiquement, ces boîtes contiennent une zone de saisie et une étiquette associée (dans la région « contenu »). Dans l'exemple ci-dessous, l'utilisateur est invité à saisir une couleur.



```

TextInputDialog boiteSaisie = new TextInputDialog("?");
boiteSaisie.setHeaderText("Choix de votre couleur préférée");
boiteSaisie.setTitle("Couleur");
boiteSaisie.setContentText("Entrez votre couleur favorite : ");

Optional<String> resultat = boiteSaisie.showAndWait();
String couleurSaisie;

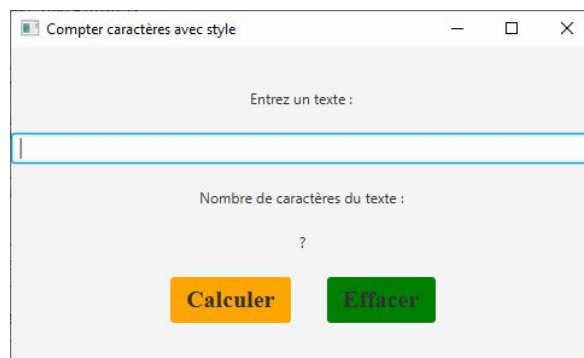
/*
 * Si l'utilisateur a cliqué sur le bouton OK :
 *     couleurSaisie reçoit la couleur
 *     ou le caractère '?' si l'utilisateur n'a rien saisi
 * Si l'utilisateur a cliqué sur le bouton Annuler ou la croix de fermeture :
 *     couleurSaisie reçoit la chaîne "aucune"
 */
couleurSaisie = resultat.map(valeur -> { return valeur; } ).orElse("aucune");

```

## 2) Les styles CSS

Il est possible d'associer un style CSS à un contrôle ou à un conteneur *layout*. Pour ce faire, on peut donner les indications de style directement dans le code Java en invoquant la méthode ***setStyle*** sur l'élément. Ou bien on peut aussi définir le style dans un fichier séparé (donc une feuille de style) que l'on applique ensuite grâce à la méthode ***getStylesheets***.

### Exemple 1 : les informations de style sont décrites dans le code Java



```

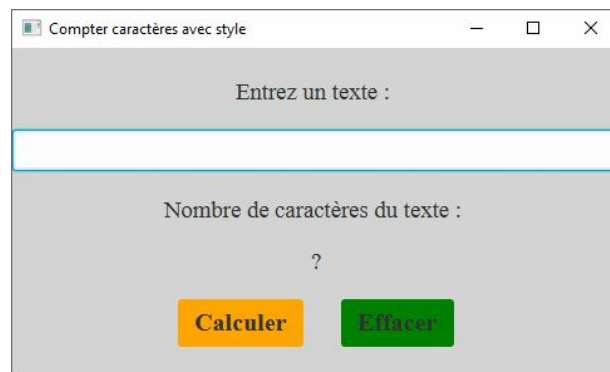
/*
 * création des 2 boutons et association d'un style css :
 * - une couleur de fond
 * - un libellé en gras de taille 20 px, avec la police serif
 */
Button calculer = new Button("Calculer");
calculer.setStyle("-fx-background-color: orange;"
    + "-fx-font: bold 20px 'serif'");

Button effacer = new Button("Effacer");
effacer.setStyle("-fx-background-color: green;"
    + "-fx-font: bold 20px 'serif'");

```

## Exemple 2 : les informations de style sont décrites dans une feuille de style

On souhaite appliquer un style à l'ensemble de la scène : modifier la couleur de fond de la fenêtre, la police de caractères et la taille des caractères :



Dans le paquetage de l'application, il faut définir un fichier nommé par exemple *stylesimple.css* et contenant le code suivant :

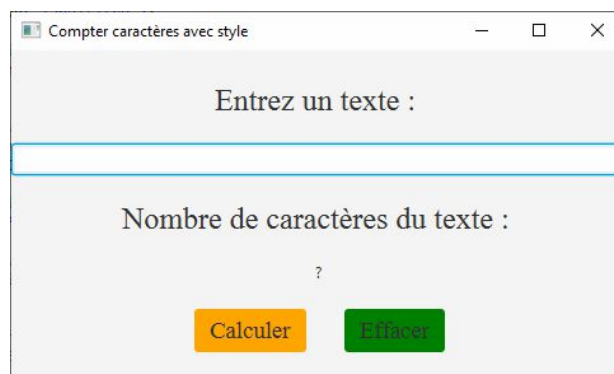
```
.root
{
    -fx-background-color : lightgray;
    -fx-font-family: "serif";
    -fx-font-size: 14pt;
}
```

Ensuite dans le code Java, on applique ce style à la scène de la manière suivante :

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

scene.getStylesheets().add(getClass().getResource("stylesimple.css").toExternalForm());
```

## Exemple 3 : dans la feuille de style, on définit le style de plusieurs composants



On peut définir dans la feuille de style, le style à appliquer à différents contrôles. On suppose que l'on a défini dans le fichier *stylecomplet.css*, les styles suivants :

```
.buttongreen{
    -fx-background-color: green;
    -fx-font-family: serif;
    -fx-font-size: 14pt;
}

.buttonorange{
    -fx-background-color: orange;
    -fx-font-family: serif;
    -fx-font-size: 14pt;
}

.labeltitre {
    -fx-font-family: serif;
    -fx-font-size: 18pt;
    -fx-font-color: red;
}
```

Dans le code Java, il faut appliquer cette feuille de style à la scène et aussi chacun des styles aux contrôles concernés :

```
// création zone de texte pour inviter l'utilisateur à saisir
Label invitation = new Label("Entrez un texte : ");
invitation.getStyleClass().add("labeltitre");

// création d'une zone de saisie
TextField zoneSaisieTexte = new TextField();

// création zone de texte pour expliquer le résultat
Label explicationResultat = new Label("Nombre de caractères du texte : ");
explicationResultat.getStyleClass().add("labeltitre");

// création zone de texte qui contiendra la valeur du résultat
Label resultat = new Label("?");

. . . . .

/*
 * création des 2 boutons et association d'un style css :
 *   - une couleur de fond
 *   - un libellé en gras de taille 20 px, avec la police serif
 */
Button calculer = new Button("Calculer");
calculer.getStyleClass().add("buttonorange");

Button effacer = new Button("Effacer");
effacer.getStyleClass().add("buttongreen");

. . . . .

// on crée une nouvelle scène
Scene scene = new Scene(racine);
scene.getStylesheets().add(getClass().getResource("stylecomplet.css").toExternalForm());
primaryStage.setScene(scene);

// on demande à ce que la fenêtre soit affichée
primaryStage.show();
```

# Langage FXML et SceneBuilder

Pour simplifier la tâche du programmeur, il est possible de décrire la partie statique d'une interface dans le langage FXML. Cette répartition du code, FXML pour décrire l'interface, Java pour coder les interactions de l'utilisateur via l'interface, a l'avantage de faciliter l'approche MVC (Modèle – Vue – Contrôleur).

Le code FXML pourrait être entièrement écrit par le programmeur. Mais il est plus simple d'utiliser un éditeur graphique (un outil de conception d'interfaces) qui génèrera automatiquement le code FXML. L'éditeur graphique qui sera présenté dans ce chapitre est *SceneBuilder*.

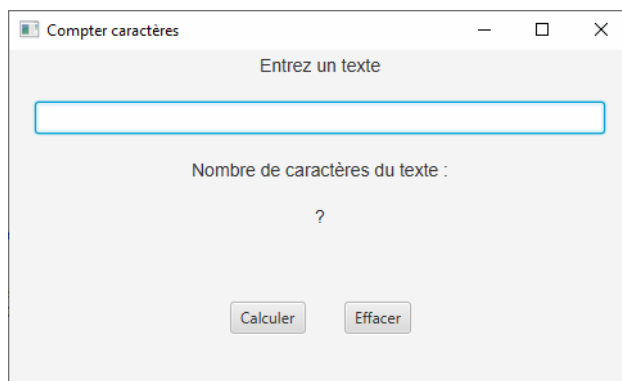
FXML est un langage de balisage dérivé de XML et pris en charge par les bibliothèques JavaFX.

## 1) Premier exemple

### 1.1) Démarche

On suppose que l'on souhaite coder l'application qui permet de compter le nombre de caractères d'une chaîne, en décrivant l'interface dans le langage FXML. Le code sera réparti en 3 fichiers :

- un fichier FXML qui décrit la partie statique de l'interface
- une classe Java qui joue le rôle du contrôleur et va gérer les événements qui se produisent sur l'interface (clics sur les boutons, récupération des saisies de l'utilisateur, affichage du résultat ...)
- une classe pour décrire la fenêtre de l'application. Comme dans les chapitres précédents, cette classe doit hériter de la classe ***Application*** et définir une méthode ***start***



La démarche du programmeur sera la suivante :

- 1) utiliser l'éditeur graphique *SceneBuilder* pour placer les *widgets* de l'interface comme il le souhaite et définir leurs caractéristiques visuelles
- 2) renseigner dans l'éditeur *SceneBuilder* des propriétés pour les *widgets* qui devront être gérés dans le code Java. Par exemple :
  - pour un bouton, nom de la méthode qui doit être appelée automatiquement lors d'un clic sur celui-ci
  - pour un *TextField*, associer un identifiant de manière à pouvoir accéder à cette zone de saisie dans le code Java, dans le but de récupérer le texte saisi par l'utilisateur
- 3) Coder la classe qui hérite de la classe *Application*
- 4) Coder une classe contrôleur pour gérer l'interaction avec l'utilisateur

## 1.2) Code FXML

Le code FXML généré par *SceneBuilder* est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- vue de l'application qui permet de compter les caractères d'une chaîne -->
<?import javafx.geometry.*?>
<?import javafx.scene.text.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>

<VBox alignment="CENTER" maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" spacing="20.0"
xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="application.ControleurVue1">

  <children>

    <Label alignment="CENTER" contentDisplay="CENTER" text="Entrez un texte"
      textAlignment="CENTER">
      <font>
        <Font name="Arial" size="14.0" />
      </font>
      <opaqueInsets>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
      </opaqueInsets>
    </Label>

    <TextField fx:id="zoneSaisie" />

    <Label text="Nombre de caractères du texte :">
      <font>
        <Font name="Arial" size="14.0" />
      </font>
    </Label>

    <Label fx:id="etiquetteResultat" text="?">
      <font>
        <Font name="Arial" size="14.0" />
      </font>
    </Label>

  </children>
</VBox>
```

```

<HBox alignment="CENTER" prefHeight="100.0" prefWidth="200.0" spacing="30.0">

    <children>
        <Button mnemonicParsing="false" onAction="#gererClicCalculer" text="Calculer" />
        <Button mnemonicParsing="false" onAction="#gererClicEffacer" text="Effacer" />
    </children>

</HBox>
</children>

<padding>
    <Insets left="20.0" right="20.0" />
</padding>

</VBox>

```

### Fichier *VueExemple1.fxml*

On observe que le fichier contient les éléments suivants :

- le type d'encodage utilisé
- les directives d'importation
- une balise racine, ici *VBOX*, qui généralement précise la nature du conteneur racine. Des propriétés sont renseignées pour ce *layout*. Vous reconnaissez par exemple, l'alignement, la propriété *spacing*
- une balise *children* qui entoure les enfants du *VBOX*
- en tant qu'enfant, on trouve : des labels, un *TextField*, et un *layout* de type *HBOX*
- ce *layout HBOX* contient les boutons

Les propriétés surlignées en jaune vont permettre d'assurer le lien avec le code Java :

- on constate que le *TextField* et l'un des labels possèdent un identifiant, propriété :  
`fx:id="un identifiant"`
- les **boutons** possèdent une propriété pour spécifier le nom de la méthode Java qui devra être invoquée automatiquement lors du clic sur le bouton :  
`onAction="#nom de la méthode"`
- une propriété associée à l'élément racine, ici le *VBox*, qui indique le nom de la classe Java qui sera le contrôleur associé à la vue décrite par le fichier FXML :  
`fx:controller="nom de la classe Java qui est le contrôleur de la vue",`

plus exactement, on indique le nom du package, puis le nom de la classe, séparés par un point .

## 1.3) Code de la classe *Application*

Dans la méthode *start* de la classe *Application*, il s'agit maintenant de charger la vue définie dans le fichier FXML pour qu'elle devienne la racine de la scène.

```

/*
 * Affiche une fenêtre JavaFX. Permettra de compter les caractères d'une chaîne
 * La vue est décrite dans un fichier fxml : VueExemple1.fxml
 * ExempleCoursSBl.java
 */
package application;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;

```

```

import javafx.stage.Stage;
import javafx.scene.Parent;
import javafx.scene.Scene;

/**
 * Cette classe est la classe principale d'une application JavaFX.
 *
 * La fenêtre est dotée d'une zone de saisie dans laquelle l'utilisateur sera invité
 * à saisir une phrase. L'application affichera ensuite, lors d'un clic sur le bouton
 * "Calculer", le nombre de caractères de cette phrase.
 * Un clic sur le bouton "Effacer" effacera les valeurs affichée et saisie.
 * DANS CETTE VERSION : la partie statique de la vue est décrite dans un fichier fxml
 *                       VueExemple1.fxml et les interactions avec l'utilisateur sont
 *                       gérées par une classe contrôleur ControleurVue1.java
 * @author C. Servières
 * @version 1.0
 */
public class ExempleCoursSB1 extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        // création d'un chargeur de code FXML
        FXMLLoader chargeurFXML = new FXMLLoader();

        /**
         * on indique au chargeur quelle est la vue fxml qu'il devra charger :
         * ici VueExemple1.fxml
         */
        chargeurFXML.setLocation(getClass().getResource("VueExemple1.fxml"));

        /**
         * création d'un objet de type parent qui est initialisé avec le résultat du
         * chargement de la vue FXML. Ou dit autrement le code écrit en FXML est transformé
         * en un objet Java
         */
        Parent racine = chargeurFXML.load();

        Scene scene = new Scene(racine);

        // on définit le titre, la hauteur et la largeur de la fenêtre
        primaryStage.setTitle("Compter caractères");
        primaryStage.setHeight(300);
        primaryStage.setWidth(500);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**
     * Programme principal
     * @param args argument non utilisé
     */
    public static void main(String[] args) {
        launch(args);
    }
}

```

On constate que l'on a codé les opérations suivantes :

- création d'une instance de type *FXMLLoader*, un objet capable de « charger » du code FXML dans le code Java
- on indique au chargeur le nom du fichier qu'il devra charger, en invoquant la méthode ***setLocation***. Par défaut, le fichier argument est recherché dans le paquetage courant. On peut aussi indiquer un chemin relatif par rapport au paquetage courant.
- on invoque la méthode ***load*** sur le chargeur. Celle-ci renvoie un résultat de type ***Parent***.

Lors du chargement du fichier FXML, son contenu est interprété et les objets Java correspondant aux nœuds *fxml* sont créés automatiquement.



## 1.4) Code de la classe contrôleur

La classe contrôleur est codée de la manière suivante :

```
/*
 * Classe contrôleur qui gère de la vue VueExemple1.fxml
 * fichier ControleurVue1.java
 */
package application;

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

/**
 * Classe contrôleur qui gère l'interactivité avec la vue décrite dans le fichier fxml
 * VueExemple1.fxml
 *
 * L'utilisateur est invité à saisir un texte. L'application affiche ensuite le nombre
 * de caractères du texte.
 * @author C. Servières
 * @version 1.0
 */
public class ControleurVue1 {

    /**
     * Explications détaillées :
     *
     * La vue est dotée d'une zone de saisie, nommée ici zoneSaisie, et d'une étiquette
     * nommée ici etiquetteResultat.
     *
     * Elle comporte également 2 boutons "Calculer" et "Effacer". La méthode gererClicCalculer
     * gère le clic sur le bouton "Calculer" et la méthode gererClicEffacer gère le clic sur
     * le bouton "Effacer"
     *
     * @FXML est une annotation du langage Java (en lien avec JavaFX).
     * Elle sert à indiquer que l'élément (ici un contrôle ou une méthode) est lié à un
     * élément déclaré dans le fichier fxml décrivant la vue que ce contrôleur gère
     */

    /** champ de saisie de la chaîne de caractères à analyser */
    @FXML
    private TextField zoneSaisie;

    /** Etiquette permettant d'afficher le nombre de caractères de la chaîne */
    @FXML
    private Label etiquetteResultat;

    /**
     * Méthode invoquée lors du clic sur le bouton Calculer
     * La méthode récupère la chaîne saisie par l'utilisateur. Elle affiche ensuite
     * dans l'étiquette le nombre de caractères de cette chaîne
     */
    @FXML
    private void gererClicCalculer() {
        // la méthode getText permet de récupérer le texte présent dans zoneSaisieTexte
        etiquetteResultat.setText(zoneSaisie.getText().length() + " caractère(s)");
    }

    /**
     * Méthode invoquée lors du clic sur le bouton Effacer
     * La méthode rétablit l'interface dans son état d'origine
     */
    @FXML
    private void gererClicEffacer() {
        zoneSaisie.setText("");
        etiquetteResultat.setText("?");
    }
}
```

**Fichier ControleurVue1.fxml**

On constate que le contrôleur contient des annotations `@FXML`. Cette annotation précède des attributs : ceux-ci vont être initialisés automatiquement grâce à l'identifiant qui a été renseigné dans le code FXML. Par exemple, l'attribut `zoneSaisie` sera initialisé automatiquement avec un objet Java résultat de la traduction du code FXML suivant :

```
<TextField fx:id="zoneSaisie" />
```

L'autre attribut, nommé `etiquetteResultat`, sera quant à lui initialisé avec un objet résultant de la transcription du code :

```
<Label fx:id="etiquetteResultat" text="?">
    <font>
        <Font name="Arial" size="14.0" />
    </font>
</Label>
```

On dit aussi que l'annotation `@FXML` provoquera une injection de code *fxml* dans le code Java.

Les annotations `@FXML` placées devant une méthode indiquent les actions à effectuer lorsque l'utilisateur interagit avec les *widgets*. Par exemple, on trouve une telle annotation devant la méthode `gererClicCalculer`. C'est cette méthode qui sera automatiquement invoquée lors du clic sur le bouton « Calculer », conformément à la propriété indiquée dans le code *fxml* :

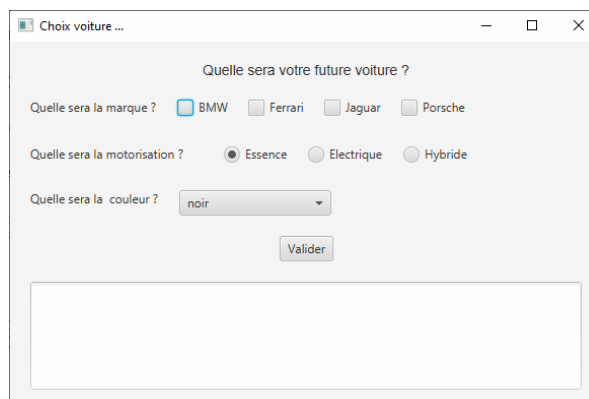
```
<Button mnemonicParsing="false" onAction="#gererClicCalculer" text="Calculer" />
```

La classe qui joue le rôle de contrôleur doit être indiquée en tant que valeur de la propriété `fx:controller`, propriété associée à l'élément racine du fichier *fxml* :

```
<VBox alignment="CENTER" maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" spacing="20.0"
xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="application.ControleurVue1">
```

## 2) Exemple 2

Pour illustrer quelques autres notions, nous allons coder l'application « Choix voiture » en utilisant du code FXML.



Dans le code FXML, il faut observer que :

- tous les contrôles auxquels on souhaitera faire accès dans le code Java (pour récupérer les saisies de l'utilisateur ou pour renseigner un résultat) possèdent une propriété `fx:id`
- le bouton « valider » possède une propriété `onAction=` qui précise le nom de la méthode Java qui sera invoquée lors du clic
- l'élément racine du fichier FXML, ici un `VBox`, possède une propriété `fx:controller` égale au nom de la classe Java qui jouera le rôle du contrôleur associé à la vue

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- vue de l'application qui permet de saisir les caractéristiques d'une voiture -->
<?import javafx.geometry.*?>
<?import javafx.scene.text.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>

<VBox alignment="TOP_CENTER" maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
    minWidth="-Infinity"
    prefHeight="400.0" prefWidth="600.0" spacing="20.0" xmlns="http://javafx.com/javafx/8"
    xmlns:fx="http://javafx.com/fxml/1" fx:controller="application.ControleurVue2">
    <children>
        <Label text="Quelle sera votre future voiture ?">
            <font>
                <Font name="Arial" size="14.0" />
            </font>
        </Label>
        <HBox prefHeight="59.0" prefWidth="560.0">
            <children>
                <Label text="Quelle sera la marque ?">
                    <padding>
                        <Insets right="20.0" />
                    </padding>
                </Label>
                <CheckBox fx:id="caseBMW" mnemonicParsing="false" text="BMW">
                    <padding>
                        <Insets right="20.0" />
                    </padding>
                </CheckBox>
                <CheckBox fx:id="caseFerrari" mnemonicParsing="false" text="Ferrari">
                    <padding>
                        <Insets right="20.0" />
                    </padding>
                </CheckBox>
                <CheckBox fx:id="caseJaguar" mnemonicParsing="false" text="Jaguar">
                    <padding>
                        <Insets right="20.0" />
                    </padding>
                </CheckBox>
                <CheckBox fx:id="casePorsche" mnemonicParsing="false" text="Porsche">
                    <padding>
                        <Insets right="20.0" />
                    </padding>
                </CheckBox>
            </children>
            <VBox.margin>
                <Insets />
            </VBox.margin>
        </HBox>
        <HBox prefHeight="57.0" prefWidth="560.0" spacing="20.0">
            <children>
                <Label text="Quelle sera la motorisation ?">
                    <padding>
```

```

        <Insets right="20.0" />
    </padding>
</Label>
<RadioButton fx:id="radioEssence" mnemonicParsing="false" selected="true" text="Essence">
    <toggleGroup>
        <ToggleGroup fx:id="GroupeMoteur" />
    </toggleGroup>
</RadioButton>
<RadioButton fx:id="radioElectrique" mnemonicParsing="false" text="Electrique">
    <toggleGroup="$GroupeMoteur" />
<RadioButton fx:id="radioHybride" mnemonicParsing="false" text="Hybride">
    <toggleGroup="$GroupeMoteur" />
</children>
</HBox>
<HBox prefHeight="57.0" prefWidth="560.0" spacing="20.0">
    <children>
        <Label text="Quelle sera la couleur ?" />
        <ComboBox fx:id="comboCouleur" prefWidth="150.0" />
    </children>
</HBox>
<Button mnemonicParsing="false" onAction="#gererBoutonValider" text="Valider" />
<TextArea fx:id="zoneResultat" prefHeight="139.0" prefWidth="560.0" />
</children>
<opaqueInsets>
    <Insets />
</opaqueInsets>
<padding>
    <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
</padding>
</VBox>

```

### Fichier VueExemple2.fxml

La méthode *start* de la classe Java qui hérite de *Application* sera codée de la manière suivante :

```

@Override
public void start(Stage primaryStage) throws Exception {

    // création d'un chargeur de code FXML
    FXMLLoader chargeurFXML = new FXMLLoader();

    /*
    * on indique au chargeur quelle est la vue fxml qu'il devra charger :
    * ici VueExemple2.fxml
    */
    chargeurFXML.setLocation(getClass().getResource("VueExemple2.fxml"));

    /*
    * création d'un objet de type parent qui est initialisé avec le résultat du
    * chargement de la vue FXML. Ou dit autrement le code écrit en FXML est transformé
    * en un objet Java
    */
    Parent racine = chargeurFXML.load();

    Scene scene = new Scene(racine);

    // on définit le titre, la hauteur et la largeur de la fenêtre
    primaryStage.setTitle("Choix voiture ...");
    primaryStage.setHeight(400);
    primaryStage.setWidth(600);
    primaryStage.setScene(scene);
    primaryStage.show();
}

```

Par rapport à l'exemple précédent, seul le nom de la vue a été modifié (ainsi que le titre de la fenêtre).

La classe contrôleur sera la suivante :

```
/*
 * Classe contrôleur qui gère de la vue VueExemple2.fxml
 * fichier ControleurVue2.java
 */
package application;

import javafx.fxml.FXML;
import javafx.scene.control.CheckBox;
import javafx.scene.control.ComboBox;
import javafx.scene.control.RadioButton;
import javafx.scene.control.TextArea;
import javafx.scene.control.ToggleGroup;

/**
 * Classe contrôleur qui gère l'interactivité avec la vue décrite dans le fichier fxml
 * VueExemple2.fxml
 *
 * L'utilisateur est invité à saisir les caractéristiques d'une voiture.
 * L'application affiche ensuite une synthèse des choix de l'utilisateur.
 *
 * @author C. Servières
 * @version 1.0
 */
public class ControleurVue2 {

    /**
     * Explications détaillées
     *
     * La vue est dotée de :
     * - 4 cases à cocher pour choisir la marque de la voiture
     * - de 3 boutons radio pour choisir la motorisation
     * - d'un groupe de boutons radio pour rassembler les 3 boutons radio
     * - d'une boîte combo pour choisir la couleur de la voiture
     * - d'une zone de texte destinée à afficher la synthèse des choix de l'utilisateur
     *
     * Elle comporte également un bouton "Valider". La méthode gererClicValider
     * gère le clic sur ce bouton.
     */

    /** Bouton radio pour le choix motorisation Electrique */
    @FXML
    private RadioButton radioElectrique;

    /** Bouton radio pour le choix motorisation Essence */
    @FXML
    private RadioButton radioEssence;

    /** Bouton radio pour le choix motorisation Hybride */
    @FXML
    private RadioButton radioHybride;

    /** Case à cocher pour le choix de la marque : Porsche */
    @FXML
    private CheckBox casePorsche;

    /** Case à cocher pour le choix de la marque : Jaguar */
    @FXML
    private CheckBox caseJaguar;

    /** Case à cocher pour le choix de la marque : Ferrari */
    @FXML
    private CheckBox caseFerrari;

    /** Case à cocher pour le choix de la marque : BMW */
    @FXML
    private CheckBox caseBMW;
```

```

/** Groupe qui rassemble tous les boutons radio
 * Le but est qu'un seul puisse être sélectionné, au sein du groupe */
@FXML
private ToggleGroup GroupeMoteur;

/** Boîte combo qui permet de choisir la couleur de la voiture */
@FXML
private ComboBox<String> comboCouleur;

/** Zone de texte (sur plusieurs lignes) qui affichera
 * toutes les caractéristiques de la voiture */
@FXML
private TextArea zoneResultat;

/**
 * Méthode appelée automatiquement lorsque la vue est chargée
 * On initialise ici les valeurs affichées par la boîte combo
 */
@FXML
private void initialize() {
    comboCouleur.getItems().addAll("noir", "blanc", "gris", "rouge", "bleu");

    // rendre le 1er élément de la liste visible directement
    comboCouleur.getSelectionModel().selectFirst();
}

/**
 * Méthode invoquée lors du clic sur le bouton Valider
 * La méthode récupère les informations renseignées et les affiche sous
 * la forme d'un texte dans le contrôle de type TextArea
 */
@FXML
private void gererBoutonValider() {
    // aAfficher est la chaîne qui sera affichée dans le TextArea
    String aAfficher = "Vous souhaitez une voiture de marque ";

    // on concatène les marques de voitures associées aux cases cochées
    if (caseBMW.isSelected()) {
        aAfficher += "BMW, ";
    }
    if (caseFerrari.isSelected()) {
        aAfficher += "Ferrari, ";
    }
    if (caseJaguar.isSelected()) {
        aAfficher += "Jaguar, ";
    }
    if (casePorsche.isSelected()) {
        aAfficher += "Porsche, ";
    }

    /*
     * Un seul bouton radio précisant la motorisation doit être sélectionné.
     * On récupère son libellé pour le concaténer à la chaîne
     */
    aAfficher += "\n\navec une motorisation ";
    if (radioEssence.isSelected()) {
        aAfficher += "essence ";
    } else if (radioElectrique.isSelected()) {
        aAfficher += "électrique ";
    } else {
        aAfficher += "hybride ";
    }

    // on récupère la couleur sélectionnée dans la ComboBox
    aAfficher += "\n\net sa couleur sera " + comboCouleur.getValue();

    // le texte constitué est affiché dans le TextArea
    zoneResultat.setText(aAfficher);
}
}

```

Par rapport à l'exemple précédent, on note la présence d'une méthode nommée *initialize*. Cette méthode est automatiquement invoquée juste après le chargement de la vue dans le code Java. Généralement, on code dans cette méthode l'initialisation de certains composants de la vue. Ici, on a initialisé la boîte combo qui ne pouvait pas l'être facilement dans le code *fxml*.

### 3) Utilisation de SceneBuilder

#### 1) Création du fichier fxml qui contiendra la vue

Une fois le projet créé avec Eclipse, il faut lui ajouter un fichier *fxml* qui contiendra la vue de l'application. Pour ce faire : clic droit sur le paquetage dans lequel on souhaite ajouter le fichier *fxml* (probablement le paquetage *application*). Puis :

New | Others... | JavaFX | New FXML Document

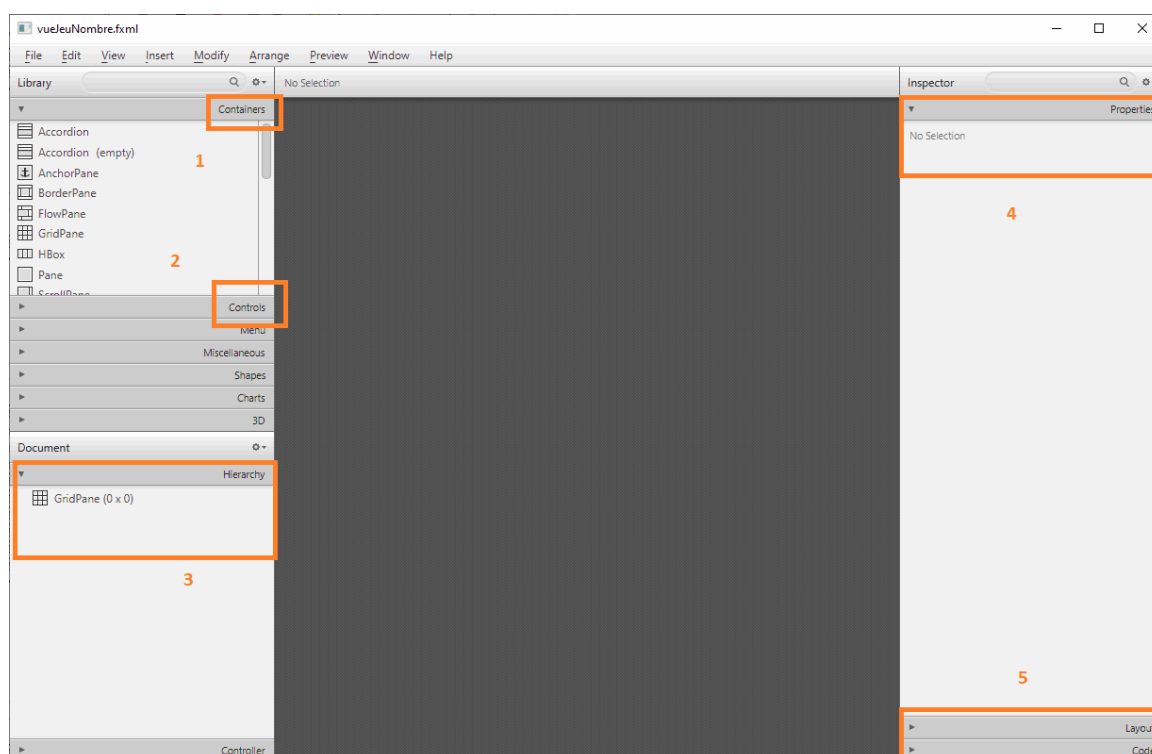
Puis cliquer sur « Next » et renseigner le nom du fichier *fxml* à créer. Eventuellement, on peut modifier l'élément *layout* qui sera situé à la racine du fichier (par défaut c'est *AnchorPane*).

Puis cliquer sur « Finish »

#### 2) Ouvrir SceneBuilder

Vous pouvez déjà visualiser sous Eclipse le contenu du fichier créé en l'ouvrant dans l'éditeur. On constate qu'il contient peu de code pour l'instant.

Pour démarrer *SceneBuilder*, faire un clic droit sur le fichier *fxml* qui contient la vue. Puis choisir « Open with SceneBuilder ». On obtient la fenêtre suivante :



On remarque 5 volets importants :

- 1) « Containers » pour choisir un element layout contenant
- 2) « Controls » pour choisir un contrôle à insérer
- 3) « Hierarchy » pour visualiser la hiérarchie des composants que nous avons déjà placés sur l'interface (pour l'instant aucun sauf le *layout GridPane*)
- 4) Dans l'inspecteur à droite, « Properties » pour voir les propriétés de l'élément (contrôle ou *layout*) actuellement sélectionné
- 5) « Layout » et « Code » pour voir d'autres propriétés de ce même éléments : celles de positionnement et celles en lien avec le code Java

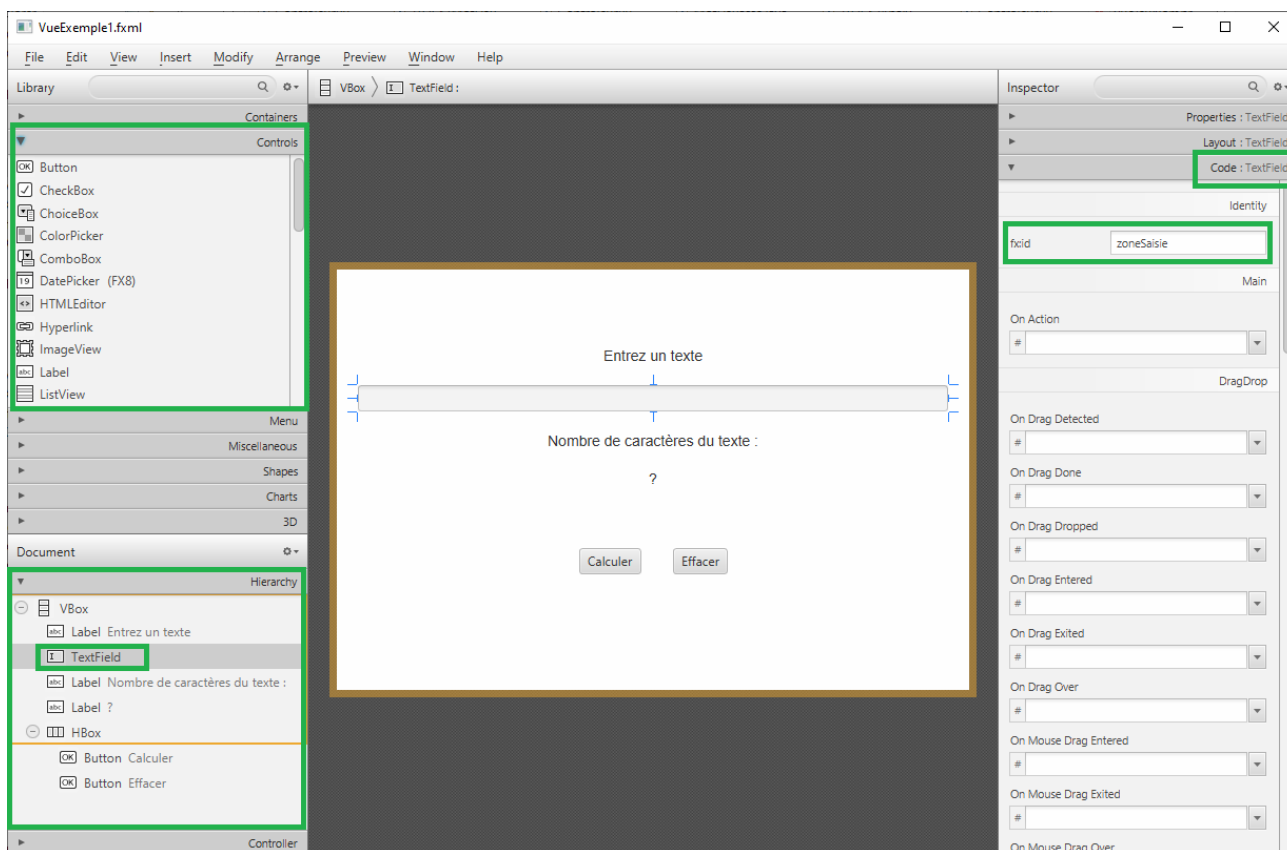
### 3) Constituer la vue à partir de SceneBuilder

La copie d'écran ci-dessous montre l'interface du 1<sup>er</sup> exemple de ce chapitre. On voit au centre de SceneBuilder l'interface en cours de construction.

On constate à gauche que le volet « Controls » est ouvert ce qui nous permettrait de placer un contrôle supplémentaire sur l'interface, par un simple glisser-déposer.

Le volet « Hierarchy » nous montre l'arborescence des composants. Par exemple, on voit que les boutons sont placés dans un *HBox*, lui-même placé dans un *VBox*. Actuellement, c'est le contrôle de type *TextField* qui est sélectionné.

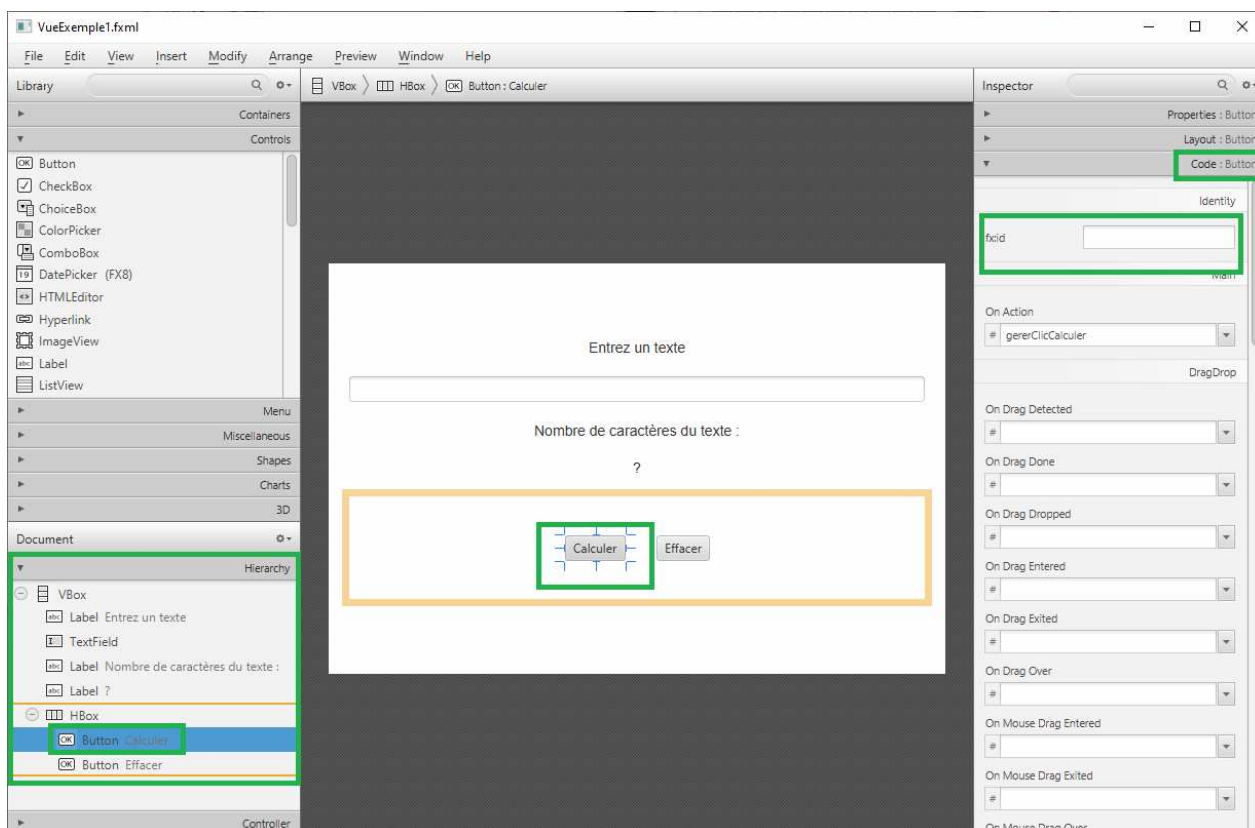
Donc dans les volets situés à droite, on peut accéder au détail de ce contrôle pour renseigner des caractéristiques. Actuellement, c'est le volet « Code » qui est ouvert. On remarque que la propriété *fx:id* est renseigné avec l'identifiant du *widget*.



Dans la copie d'écran ci-dessous, c'est le contrôle bouton « calculer » qui est sélectionné dans la hiérarchie. Dans les volets de l'inspecteur à droite, on constate que le bouton n'a pas d'identifiant, mais



que par contre (juste en dessous), on lui a associé un nom de méthode « gererClicCalculer », méthode qui sera invoquée lors du clic sur ce bouton.



#### **4) Renseigner la propriété fx :contrôler**

Le programmeur doit placer directement cette propriété dans le fichier *fxml* décrivant la vue (et pas à partir de SceneBuilder)

#### **5) Générer une partie du code de la classe contrôleur**

Pour ce faire dans la barre de menu de SceneBuilder :

View | Show Sample Controller Skeleton