UMONS
Université de Mons

Java FX
Introduction

R. Absil

Faculté des Sciences

Année académique 2020 - 2021

Dans ce document vous trouverez une introduction sur l'utilisation de *JavaFX*, une librairie standard pour le développement des interfaces graphiques en Java.

Table des matières

1 Hello World

Afin de commencer l'écriture d'une interface graphique *JavaFX*, on présente un programme `HelloWorld` classique, sans se soucier de l'ensemble des détails d'écriture.

Cet exemple, illustré à la figure 1, est composé d'une fenêtre simple munie d'un unique champs de texte « Hello World ». Notez que le *look and feel* dépend de l'OS sur lequel il est exécuté : ici, une distribution Ubuntu 20.10.

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.text.Text;
4  import javafx.scene.layout.BorderPane;
5  import javafx.stage.Stage;
6
7  import static javafx.application.Application.launch;
8
9  public class HelloWorld extends Application
10 {
11
12     public static void main(String[] args)
13     {
14         launch(args);
15     }
```

```
16
17      @Override
18      public void start(Stage primaryStage)
19      {
20          primaryStage.setTitle("My First JavaFX App");
21
22          BorderPane root = new BorderPane();
23          Text helloText = new Text("Hello World");
24          root.setCenter(helloText);
25          Scene scene = new Scene(root, 250, 100);
26          primaryStage.setScene(scene);
27          primaryStage.show();
28      }
29  }
```

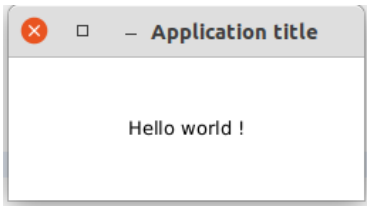


FIGURE 1 – Résultat de l'exécution du `HelloWorld`

De cet exemple et de la documentation [1] de la classe `Application`, on remarque les choses suivantes :

JavaFX : la classe `Application`

Une application *JavaFX* hérite de la classe `Application`. Pour lancer une telle application il faut exécuter la méthode statique`Application.launch()`.
Cette méthode

- appelle la méthode d'initialisation `Application.init()`,
- appelle la méthode `Application.start()`, point d'entrée de l'application, elle prend en paramètre une instance de la scène principale : `start(Stage primaryStage)`,
- attend que l'application se termine [a] ;
- appelle la méthode `Application.stop()` qui se charge de clôturer l'application.

a. On reviendra plus tard sur comment le système détermine la fin d'une application.

2 La salle de spectacle

La description d'une application JavaFX suit l'analogie de la salle de spectacle. On y trouve une `Stage` qui constitue l'endroit où se déroule le spectacle (`Scene`).

```
1  import javafx.stage.Stage;
2  import javafx.scene.Scene;
```

Au sein de cette `Scene` on trouve un premier `Layout` dont le but est d'organiser le positionnement des composants de l'interface graphique. On détaille ce concept à la section 4.

```
1  import javafx.scene.layout.BorderPane
```

1. https://openjfx.io/javadoc/13/javafx.graphics/javafx/application/Application.html

Le layout `BorderPane`, illustré à la figure 2, divise l'écran en cinq zones : *Top*, *Bottom*, *Left*, *Right* et *Center*.

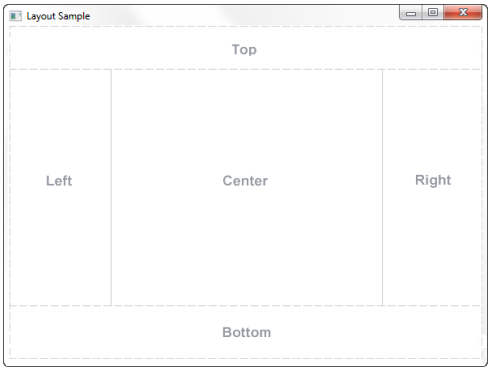


FIGURE 2 – Le *layout* `BorderPane` divise un écran en 5 zones

Finalement, dans une des zones du `Layout`, on trouve les composants de l'interface graphique. Il s'agit des libellés, boutons ou autres boîtes à cocher constituant les éléments actifs de l'application. Ils sont appelés `Controls`.

Chaque composant nécessite une importation spécifique. Dans cet exemple, on utilise un libéllé textuel, et il est nécessaire d'effectuer l'import suivant :

```
1 import javafx.scene.text.Text
```

3 Les composants, controls, widgets

Une interface graphique passe par l'utilisation de boutons, de libellés, de zone de texte, de cases à cocher et de différents objets appelés *composants*[2].

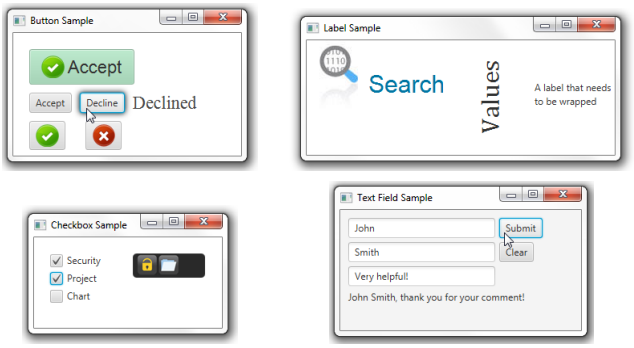


TABLE 1 – Différents composants JavaFX : Bouton, Label, CheckBox et TextField

Vous trouverez des tutoriels sur les différents composants à l'adresse suivante[3]. Le but des

2. Ces composants d'interface sont fréquemment nommés *controls* dans la documentation en anglais
3. http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm

prochaines sections est de vous familiariser avec certains composants et de vous inviter à lire la documentation qui les concernent.

Pour comprendre le fonctionnement des composants non explorés dans ce document, il faudra vous baser uniquement sur la documentation. Notez que la liste des composants s'enrichit chaque jour, notamment par des projets tel que ControlsFX [4].

3.1 Labeled

De nombreux composants affichent et gèrent des textes (libellés, boutons, cases à cocher, etc). Afin de structurer au mieux le code, les différents composants gérant du texte vont hériter de la classe `Labeled`, comme vous pouvez le voir sur la figure 3. Pour des raisons de présentation, l'ensemble des composants bénéficiant de cet héritage ne sont pas repris sur la figure.

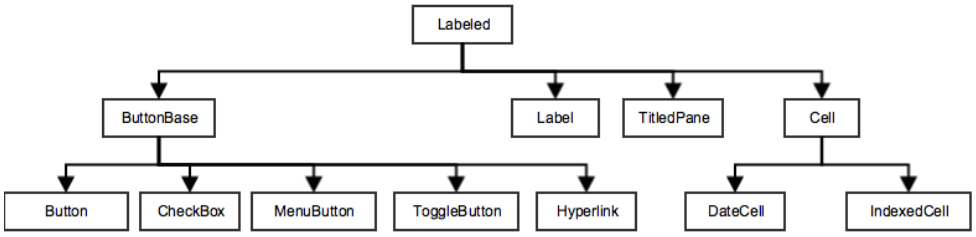


FIGURE 3 – Les composants héritant de `Labeled` peuvent gérer du texte.

Propriétés

Afin de décrire un composant on lui associe un ensemble de *propriétés*. Celles-ci sont des attributs de notre composant :
- on accède aux propriétés d'un composant via un getter ;
- on modifie les propriétés d'un composant via un setter.

Dans le cas d'un composant `Labeled`, voici trois propriétés intéressantes :
— le texte affiché : `text` ;
— la police : `font` ;
— la couleur du texte : `textFill`.

Pour obtenir la liste des propriétés d'un composant `Labeled`, vous pouvez consulter la javadoc du composant [5].

3.1.1 Label

Le composant `Label` représente un texte non éditable, un libellé. Ce composant hérite de `Labeled` et de ses propriétés. On peut modifier le programme `HelloWorld` de la section 1 pour utiliser ces propriétés.

4. http://fxexperience.com/controlsfx/features/
5. https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/control/Labeled.html

```java
package esi.atl.fx.hello;

//import ...

public class HelloWorldProperty
        extends Application {

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("My First JavaFX App");
        BorderPane root = new BorderPane();
        Label helloText = new Label("Hello World");
        helloText.setTextFill(Color.RED);
        helloText.setFont(Font.font("Verdana", 20));

        System.out.println("Le message du Libellé est " + helloText.getText());
        System.out.println("La police du Libellé est " + helloText.getFont());
        System.out.println("La couleur du Libellé est " + helloText.getTextFill());

        root.setCenter(helloText);
        Scene scene = new Scene(root, 250, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

}
```

Comme le stipule la documentation [6], plusieurs constructeurs sont disponibles pour le composant `Label`. Dans l'exemple ci-dessus, on utilise celui qui prend en paramètre le texte à afficher à l'écran. Notez également la manière d'accéder à la propriété `TextFill` du composant `helloText`.

3.1.2 Check Box

Ce composant est typiquement utilisé lorsque l'utilisateur peut choisir parmi plusieurs options qui peuvent être simultanément activées.

Le composant `CheckBox` représente une case à cocher qui est caractérisée par trois états : Désélectionné, Sélectionné, Indéterminé, tel qu'illustré à la figure 4. Chaque clic de l'utilisateur fera passer le composant d'un état à un autre.

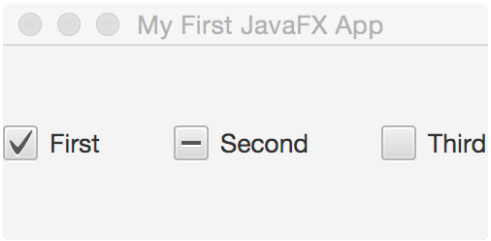


FIGURE 4 – Résultat de l'exécution du `HelloWorldCheckBox`. Chaque box est dans un état différent

Comme expliqué dans la javadoc [7] cette classe dispose de deux constructeurs :
— `new CheckBox()` : une `CheckBox` sans libellé ;
— `new CheckBox("Faites votre choix")` : une `CheckBox` avec libellé.
Les propriétés non héritées d'une `CheckBox` sont :

6. https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/control/Label.html
7. https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/control/CheckBox.html

— `allowIndeterminate` : si l'attribut prend la valeur `false`, la check box passe par deux états (sélectionné, déselectionné), dans le cas contraire on ajoute un troisième état (indeterminé) ;
— `indeterminate` : indique si l'état de la case est indeterminé ;
— `selected` : indique si la case est sélectionnée.

```
1  package esi.atl.fx.hello;
2  //import ...
3
4  public class HelloWorldCheckBox
5          extends Application {
6
7      public static void main(String[] args) {
8          launch(args);
9      }
10
11     @Override
12     public void start(Stage primaryStage) {
13         primaryStage.setTitle("My First JavaFX App");
14         BorderPane root = new BorderPane();
15
16         CheckBox checkBox1 = new CheckBox();
17         checkBox1.setText("First");
18         checkBox1.setSelected(true);
19
20         CheckBox checkBox2 = new CheckBox("Second");
21         checkBox2.setIndeterminate(true);
22
23         CheckBox checkBox3 = new CheckBox("Third");
24
25         checkBox3.setAllowIndeterminate(true);
26
27         //Alignment
28         root.setLeft(checkBox1);
29         BorderPane.setAlignment(checkBox1, Pos.CENTER);
30         root.setCenter(checkBox2);
31         root.setRight(checkBox3);
32         BorderPane.setAlignment(checkBox3, Pos.CENTER);
33
34         Scene scene = new Scene(root, 250, 100);
35         primaryStage.setScene(scene);
36         primaryStage.show();
37     }
38 }
```

3.2 TextInputControl

Comme on peut le constater sur la figure 5, la classe abstraite `TextInputControl`[8] est la classe parente de différents composants qui permettent à l'utilisateur de saisir du texte.

Il s'agit notamment des composants d'interface tels que `TextField`, `PasswordField` et `TextArea`.

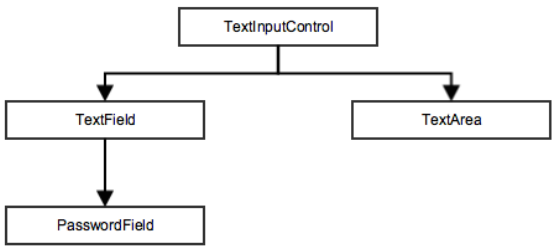


FIGURE 5 – La classe abstraite `TextInputControl` est la classe parente des composants `TextField`, `PasswordField` et `TextArea`.

8. https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/control/TextInputControl.html

Cette classe définit les propriétés de base et les fonctionnalités communes aux composants offrant une saisie de texte.

Ces propriétés permettent la sélection et l'édition de texte ou la gestion du curseur à l'intérieur du texte (*caret*). On peut citer les propriétés :
- `text` : le texte contenu dans le composant ;
- `editable` : booléen rendant le texte éditable par l'utilisateur ;
- `font` : la police du texte ;
- `length` : la longueur du texte.

Ainsi que les méthodes :
- `clear()` : efface le texte du composant ;
- `insertText()` : insère une chaine de caractères dans le texte ;
- `appendText()` : ajoute une chaine de caractères à la fin du texte ;
- `selectAll()` : sélectionne l'ensemble du texte.

3.2.1 TextField

Le composant `TextField`[9] représente un champ texte d'une seule ligne qui est éditable par défaut. Il peut également être utilisé pour afficher du texte.

Ce composant hérite des propriétés de `TextInputControl` mais possède également les propriétés suivantes :
- `alignment` : défini l'alignement du texte ;
- `prefColumnCount` : définit le nombre de colonnes pour ce champs texte. La valeur par défaut est de 12 ;
- `onAction` : définit un événement à générer lors d'une certaine action de l'utilisateur, par défaut, lorsque l'utilisateur presse la touche *enter*.

Un exemple d'utilisation des deux premières propriétés est décrit dans l'exemple ci-dessous.

```java
package esi.atl.fx.hello;
//import ...
public class HelloWorldTextField extends Application {

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("My First JavaFX App");
        BorderPane root = new BorderPane();

        Label userName = new Label("User Name");

        TextField tfdUserName = new TextField();
        tfdUserName.setPrefColumnCount(12);
        tfdUserName.setAlignment(Pos.CENTER_LEFT);

        //Alignment
        root.setTop(userName);
        BorderPane.setAlignment(userName, Pos.CENTER);
        root.setCenter(tfdUserName);

        Scene scene = new Scene(root, 250, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

9. https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/control/TextField.html

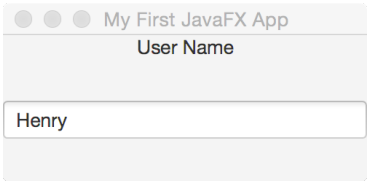


FIGURE 6 – Résultat de l'exécution du `HelloWorldtextField`. Le texte est aligné sur la gauche.

3.2.2 TextArea

Le composant `TextArea` [10] permet d'afficher et de saisir du texte dans un champ multilignes (une zone de texte).

Le texte peut être renvoyé à la ligne automatiquement (wrapping) et des barres de défilement (scrollbar) horizontales et/ou verticales sont ajoutées automatiquement si la taille du composant ne permet pas d'afficher l'entiereté du texte. Tous les caractères du texte possèdent les mêmes attributs (police, style, taille, couleur, ...).

FIGURE 7 – Résultat de l'exécution du `HelloWorldTextArea`

.

4 Les Layouts

Jusqu'à présent le positionnement des composants au sein d'un écran s'est fait en utilisant les méthodes `setCenter()` ou `setAlignment()` d'un objet `BorderPane`. Celui-ci fait partie de la famille des `layouts` [11].

Ces layouts permettent de définir la position au sein d'une fenêtre des différents composants, les positions relatives de chaque composant ainsi que les alignements et espacements de ceux-ci.

A moins que ce ne soit spécifié explicitement par le programmeur la disposition des composants est déléguée à des gestionnaires de disposition (*layout managers*) qui sont associés à ces conteneurs.

10. https://openjfx.io/javadoc/13/javafx.controls/javafx/scene/control/TextArea.html
11. Appelés également panes, layout-panes ou conteneurs

4.1 Le graphe de scène

Le graphe de scène (*scene graph*), illustré à la figure 8 est une notion importante qui représente la structure hiérarchique de l'interface graphique.

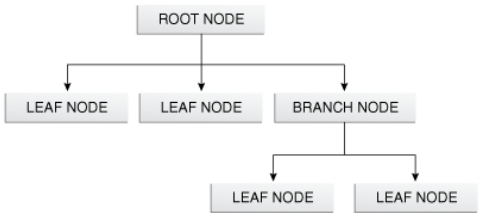


FIGURE 8 – Représentation d'un graphe de scène

Il s'agit d'un arbre orienté constitué d'un nœud origine (root) et de nœuds enfants. Tous les éléments contenus dans un graphe de scène sont des objets qui ont pour classe parente la classe `Node`.

Les *layouts* et les composants héritent tous de cette classe. Cette représentation en terme d'arbre justifie l'utilisation de la méthode `node.getChildren()` avant l'ajout d'un composant à un *layout*.

4.2 Les différents Layouts

Cette section présente une partie des différents layouts à la disposition du développeur.

4.2.1 BorderPane

Comme illustré en début de ce document, le conteneur `BorderPane` est divisé en cinq zones qui peuvent chacune contenir un seul objet `Node` : *Top*, *Bottom*, *Left*, *Right* et *Center*.

Cette situation est illustrée à la figure 9.

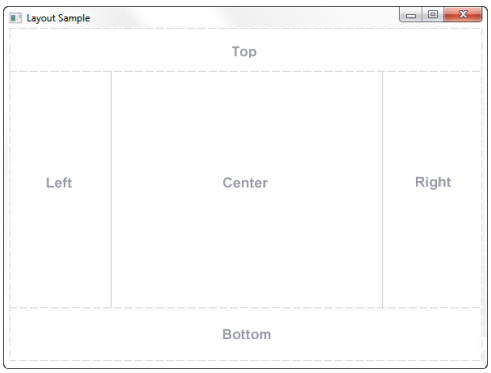


FIGURE 9 – Le *layout* `BorderPane` divise un écran en 5 zones

On remarque que
— les composants placés dans les zones *Top* et *Bottom* conservent leurs hauteurs préférées (`prefHeightProperty`) et sont éventuellement agrandis ou réduits horizontalement en fonction de la largeur du conteneur ;
— les composants placés dans les zones *Left* et *Right* conservent leurs largeurs préférées (`prefWidthProperty`) et sont éventuellement agrandis ou réduits verticalement ;
— le composant placé dans la zone *Center* est éventuellement redimensionné pour occuper l'espace restant au centre du conteneur ;
— si aucun composant n'est ajouté à une zone, celle-ci n'occupe plus aucun espace.

N'oubliez pas que pour gérer la position d'un nœud au sein d'une zone, les méthodes suivantes sont disponibles :
— `alignment()` : permet de modifier l'alignement par défaut du composant passé en paramètre ;
— `margin()` : fixe une marge (objet de type `Insets`) autour du composant passé en paramètre.

4.2.2 HBox, VBox et FlowPane

Les layouts HBox, VBox et FLowPane permettent de disposer facilement des composants en ligne ou en colonne. HBox les place horizontalement, VBox verticalement, et FlowPane « passe à la ligne » lorsqu'il n'y a plus de place disponible.

Le *layout* HBox [12] place les composants sur une ligne horizontale de gauche à droite à la suite les uns des autres.

La classe `HBoxSample` propose un exemple de répartition de trois composants `CheckBox`, illustré à la figure 10. Remarquez l'utilisation de la méthode `getChildren()` afin d'ajouter les composants au *layout*.

```
1  package esi.atl.fx.layout;
2
3  //import ...
4
5  public class HBoxSample extends Application {
6
7      public static void main(String[] args) {
8          launch(args);
9      }
10
11     @Override
12     public void start(Stage primaryStage) throws Exception {
13         primaryStage.setTitle("My First JavaFX App");
14         HBox root = new HBox(10);
15         root.setAlignment(Pos.CENTER);
16
17         CheckBox checkBox1 = new CheckBox();
18         checkBox1.setText("First");
19         checkBox1.setSelected(true);
20
21         CheckBox checkBox2 = new CheckBox("Second");
22         checkBox2.setIndeterminate(true);
23
24         CheckBox checkBox3 = new CheckBox("Third");
25
26         checkBox3.setAllowIndeterminate(true);
27
28         root.getChildren().add(checkBox1);
29         root.getChildren().add(checkBox2);
30         root.getChildren().add(checkBox3);
31
32         Scene scene = new Scene(root, 250, 100);
```

12. https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/layout/HBox.html

```
33          primaryStage.setScene(scene);
34          primaryStage.show();
35      }
36
37  }
```

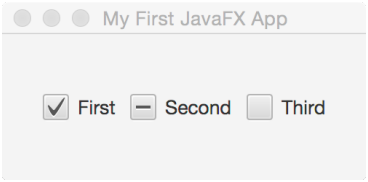


FIGURE 10 – Résultat de l'exécution du `HBoxSample`.

Le *layout* VBox[13] place les composants sur une ligne verticale de haut en bas à la suite les uns des autres.

La classe `VBoxSample` propose un exemple de répartition de trois composants `CheckBox`, illustré à la figure 11. Remarquez l'utilisation de la méthode `getChildren()` afin d'ajouter les composants au *layout*.

```
1   package esi.atl.fx.layout;
2
3   //import ...
4
5   public class VBoxSample extends Application {
6
7       public static void main(String[] args) {
8           launch(args);
9       }
10
11      @Override
12      public void start(Stage primaryStage) throws Exception {
13          primaryStage.setTitle("My First JavaFX App");
14          VBox root = new VBox(10);
15          root.setAlignment(Pos.CENTER_LEFT);
16
17          CheckBox checkBox1 = new CheckBox();
18          checkBox1.setText("First");
19          checkBox1.setSelected(true);
20
21          CheckBox checkBox2 = new CheckBox("Second");
22          checkBox2.setIndeterminate(true);
23
24          CheckBox checkBox3 = new CheckBox("Third");
25
26          checkBox3.setAllowIndeterminate(true);
27
28          root.getChildren().addAll(checkBox1,checkBox2,checkBox3);
29
30          Scene scene = new Scene(root, 250, 100);
31          primaryStage.setScene(scene);
32          primaryStage.show();
33      }
34
35  }
```

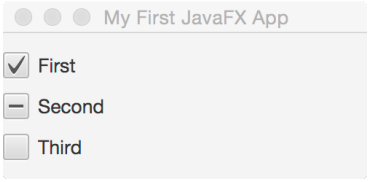


FIGURE 11 – Résultat de l'exécution du `VBoxSample`.

13. https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/layout/VBox.html

Le *layout* `FlowPane` [14] place les composants sur une ligne horizontale ou verticale et passe à la ligne ou à la colonne suivante (wrapping) lorsqu'il n'y a plus assez de place disponible. Le résultat de ce layout est illustré à la figure 12. Agrandissez la fenêtre pour observer les modifications d'affichages des différents libellés.

Comme on peut le constater dans le code de la classe `FlowPaneSample`, l'orientation du *layout* peut se décider par le passage d'un paramètre de type `Orientation` au constructeur.

```
1  package esi.atl.fx.layout;
2
3  //import ...
4
5  import static javafx.application.Application.launch;
6
7
8  public class FlowPaneSample extends Application {
9
10     public static void main(String[] args) {
11         launch(args);
12     }
13
14     @Override
15     public void start(Stage primaryStage) throws Exception {
16         primaryStage.setTitle("My First JavaFX App");
17         FlowPane root = new FlowPane();
18         root.setAlignment(Pos.CENTER_LEFT);
19         root.setPadding(new Insets(10));
20         root.setHgap(10);
21         root.setVgap(5);
22
23         List <Label> node = new ArrayList<>() ;
24         for (int i = 0; i < 10; i++) {
25             node.add(new Label("Label Num "+i));
26         }
27
28         root.getChildren().addAll(node);
29
30         Scene scene = new Scene(root, 250, 100);
31         primaryStage.setScene(scene);
32         primaryStage.show();
33     }
34
35 }
```

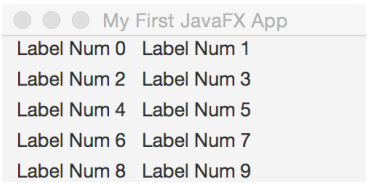


FIGURE 12 – Résultat de l'exécution du `FlowPaneSample`.

4.2.3 GridPane

Le conteneur `GridPane` [15] permet de disposer les différents composants au sein d'une grille.

Cette grille peut être irrégulière, les dimensions de ses cases ne sont pas nécessairement uniformes. La zone occupée par un composant peut s'étendre (*span*) sur plusieurs lignes et/ou colonnes. Lors de la construction d'un `GridPane` il est inutile de spécifier les nombres de lignes et de colonnes attendus, ceux-ci sont déterminés par les endroits où sont placés les composants.

14. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/FlowPane.html
15. https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/layout/GridPane.html

De même, la taille des cases de la grille est déterminée par la taille des composants les plus imposants au sein d'une ligne ou d'une colonne. Comme le montre la classe `GridPaneSample` la méthode `add()` d'un `GridPane` prend en paramètres la position de chaque composant.

```
1  package esi.atl.fx.layout;
2
3  //import ...
4
5  public class GridPaneSample extends Application {
6
7      public static void main(String[] args) {
8          launch(args);
9      }
10
11     @Override
12     public void start(Stage primaryStage) throws Exception {
13         primaryStage.setTitle("My First JavaFX App");
14         GridPane root = new GridPane();
15         root.setPadding(new Insets(10));
16         root.setHgap(10);
17         root.setVgap(5);
18
19         Label lblTitle = new Label("JavaFX Course Login");
20         lblTitle.setFont(Font.font("System", FontWeight.BOLD, 20));
21         lblTitle.setTextFill(Color.RED);
22         root.add(lblTitle, 0, 0, 2, 1);
23         GridPane.setHalignment(lblTitle, HPos.CENTER);
24         GridPane.setMargin(lblTitle, new Insets(0, 0, 10, 0));
25
26         Label lblUserName = new Label("User Name or email");
27         GridPane.setHalignment(lblUserName, HPos.RIGHT);
28         root.add(lblUserName, 0, 1);
29
30         TextField tfdUserName = new TextField();
31         tfdUserName.setPrefColumnCount(20);
32         root.add(tfdUserName, 1, 1);
33
34         Label lblPassword = new Label("Password");
35         root.add(lblPassword, 0, 2);
36
37         TextField tfdPassword = new TextField();
38         tfdPassword.setPrefColumnCount(12);
39         root.add(tfdPassword, 1, 2);
40
41         GridPane.setHalignment(lblPassword, HPos.RIGHT);
42         GridPane.setFillWidth(tfdPassword, false);
43         Scene scene = new Scene(root);
44         primaryStage.setScene(scene);
45         primaryStage.show();
46     }
47 }
```

L'utilisation d'un tel layout est illustrée à la figure 13.

FIGURE 13 – Résultat de l'exécution du `GridPaneSample`

4.2.4 Imbrication de *Layouts* : GridPane et Hbox

Les alignements proposés par l'utilisation d'un unique *layout* ne suffisent pas toujours pour réaliser l'écran imaginé par le développeur. Comme le montre la classe `MixSample` il est très fréquent d'imbriquer plusieurs conteneurs pour obtenir la disposition désirée des composants de l'interface.

```
1  package esi.atl.fx.layout;
2
3  //import ...
4
```

```
5  public class MixSample
6          extends Application {
7
8      public static void main(String[] args) {
9          launch(args);
10     }
11
12     @Override
13     public void start(Stage primaryStage) throws Exception {
14         primaryStage.setTitle("My First JavaFX App");
15         primaryStage.setMinWidth(300);
16         primaryStage.setMinHeight(200);
17         GridPane root = new GridPane();
18
19         root.setAlignment(Pos.CENTER);
20         root.setPadding(new Insets(20));
21         root.setHgap(10);
22         root.setVgap(15);
23
24         Label lblTitle = new Label("JavaFX Course Login");
25         lblTitle.setFont(Font.font("System", FontWeight.BOLD, 20));
26         lblTitle.setTextFill(Color.RED);
27         root.add(lblTitle, 0, 0, 2, 1);
28         GridPane.setHalignment(lblTitle, HPos.CENTER);
29         GridPane.setMargin(lblTitle, new Insets(0, 0, 10, 0));
30
31         Label lblUserName = new Label("User Name or email");
32         GridPane.setHalignment(lblUserName, HPos.RIGHT);
33         root.add(lblUserName, 0, 1);
34
35         TextField tfdUserName = new TextField();
36         tfdUserName.setPrefColumnCount(20);
37         root.add(tfdUserName, 1, 1);
38
39         Label lblPassword = new Label("Password");
40         root.add(lblPassword, 0, 2);
41
42         PasswordField pwfPassword = new PasswordField();
43         pwfPassword.setPrefColumnCount(12);
44         root.add(pwfPassword, 1, 2);
45
46         GridPane.setHalignment(lblPassword, HPos.RIGHT);
47         GridPane.setFillWidth(pwfPassword, false);
48
49         HBox btnPanel = new HBox(12);
50
51         Button btnLogin = new Button("Login");
52         Button btnCancel = new Button("Cancel");
53
54         btnPanel.getChildren().addAll(btnLogin,btnCancel);
55         btnPanel.setAlignment(Pos.CENTER_RIGHT);
56         root.add(btnPanel,1,3);
57         GridPane.setMargin(btnPanel, new Insets(10, 0, 0, 0));
58
59         Scene scene = new Scene(root);
60         primaryStage.setScene(scene);
61         primaryStage.show();
62     }
63
64 }
```

Le résultat de ce code est illustré à la figure 14.

FIGURE 14 – Résultat de l'exécution du `MixSample`

4.3 Autres *Layouts* disponibles

Au delà des différents *layouts* présentés dans ce document, on trouve dans la librairie *JavaFX* d'autre *layouts* :

- `StackPane` : empile les composants enfants les uns au dessus des autres dans l'ordre d'insertion ;
- `TilePane` : place les composants dans une grille alimentée soit horizontalement (par ligne, de gauche à droite) soit verticalement (par colonne, de haut en bas) ;
- `AnchorPane` : permet de positionner (ancrer) les composants enfants à une certaine distance des cotés du conteneur.

Des exemples d'utilisation de tous ces *layouts* sont disponibles sur le tutoriel [16] oracle.

5 Événement, action et gestionnaire d'événement

Les composants disposés sur un écran, il faut maintenant s'interroger sur la manière dont l'utilisateur va interagir avec ceux-ci.

Par exemple, la classe `PrintText`, illustrée à la figure 15, offre un écran qui permet d'ajouter au sein d'une zone prédéfinie le texte proposé par l'utilisateur. Si l'utilisateur appuie sur le bouton `Insert`, on souhaite que le texte entré dans le composant `TextField` apparaisse au sein du composant `TextArea`.

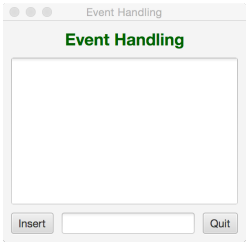


FIGURE 15 – Illustration de `PrintText`

```java
public class PrintText
        extends Application {

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

    private final BorderPane root = new BorderPane();

    private final HBox btnPanel = new HBox(10);
    private final Label lblTitle = new Label("Event Handling");
    private final TextArea txaMsg = new TextArea();
    private final Button btnInsert = new Button("Insert");
    private final TextField tfdCharacter = new TextField();
    private final Button btnQuit = new Button("Quit");
    @Override
    public void start(Stage primaryStage) throws Exception {

        primaryStage.setTitle("Event Handling");
        root.setPadding(new Insets(10));
        //---- Title
        lblTitle.setFont(Font.font("System", FontWeight.BOLD, 20));
        lblTitle.setTextFill(Color.DARKGREEN);
        BorderPane.setAlignment(lblTitle, Pos.CENTER);
        BorderPane.setMargin(lblTitle, new Insets(0, 0, 10, 0));
        root.setTop(lblTitle);
```

16. http://docs.oracle.com/javase/8/javafx/layout-tutorial/builtin_layouts.htm

```
27          //---- Text-Area
28          txaMsg.setWrapText(true);
29          txaMsg.setPrefColumnCount(15);
30          txaMsg.setPrefRowCount(10);
31          root.setCenter(txaMsg);
32          //---- Button Panel
33          btnPanel.getChildren().add(btnInsert);
34          btnPanel.getChildren().add(tfdCharacter);
35          btnPanel.getChildren().add(btnQuit);
36          btnPanel.setAlignment(Pos.CENTER_RIGHT);
37          btnPanel.setPadding(new Insets(10, 0, 0, 0));
38          root.setBottom(btnPanel);
39
40          Scene scene = new Scene(root);
41          primaryStage.setScene(scene);
42          primaryStage.show();
43
44      }
45  }
```

Si l'utilisateur clique sur le bouton `Insert`, un événement doit être déclenché.

Un événement constitue une notification qui signale que quelque chose s'est passé. Il peut être provoqué par une action de l'utilisateur, comme un clic avec la souris, une pression sur une touche du clavier ou le déplacement d'une fenêtre mais aussi par la mise à jour d'un attribut, un *timer* arrivé à échéance, une information arrivée par le réseau, etc.

En *JavaFX* les événements sont représentés par des objets de la classe `Event` [17].

Chaque instance de `Event` contient les informations suivantes :
- `EventType` : un événement est d'un certain type. L'ensemble de ces types forment une hiérarchie, illustrée à la figure 16. Si une touche du clavier est pressée, un événement de type `KeyEvent.KEY_PRESSED` est généré. Ce type a pour parent le type `KeyEvent.ANY` qui englobe les différents événements liés au clavier (`KeyEvent.KEY_RELEASED`, `KeyEvent.KEY_TYPED`,...) ;
- la source : il s'agit de l'objet qui est à l'origine de l'événement ;
- la cible (`Target`) : la cible de l'événement.

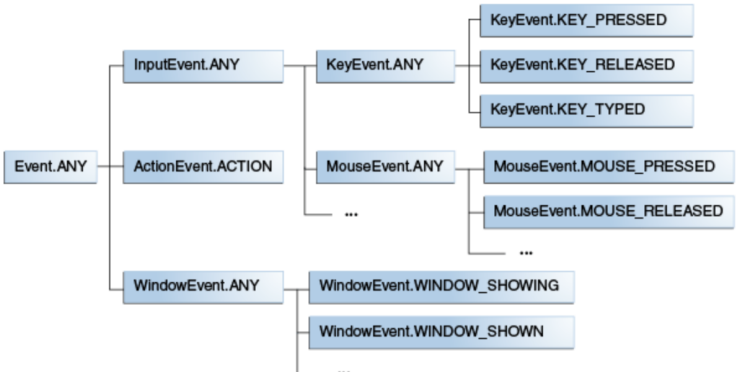


FIGURE 16 – Hiérarchie des types d'événements.

Une fois généré, cet événement va se propager le long du chemin correspondant à la chaîne de traitement (`Event Dispatch Chain`), depuis la racine (`Stage`) jusqu'à la cible (`Target`). L'événement pourra alors être arrêté via des filtres (`Event Filter`) associés aux nœuds qu'il parcourt.

Par exemple, lorsque l'utilisateur appuie sur le bouton `Insert` de la classe `PrintText` l'événement va se propager dans le graphe de scène en partant de la racine du graphe, le `Stage`, vers

17. https://openjfx.io/javadoc/13/javafx.base/javafx/event/Event.html

la cible de l'événement, le bouton `Insert`, tel qu'illustré à la figure 17.

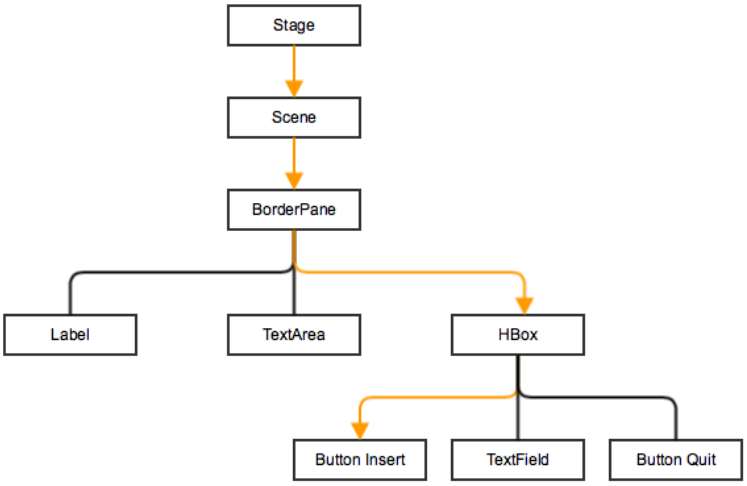


FIGURE 17 – Illustration de l'`event dispatch chain` en phase descendante

Si aucun filtre ne l'arrête, l'événement remonte ensuite depuis la cible jusqu'à la racine et les gestionnaires d'événement (`Event Listener`) sont exécutés dans l'ordre de passage. A moins qu'il ne soit consommé, un événement dont le traitement est terminé par un nœud est passé au nœud suivant de la chaîne de traitement, tel qu'illustré à la figure 18.

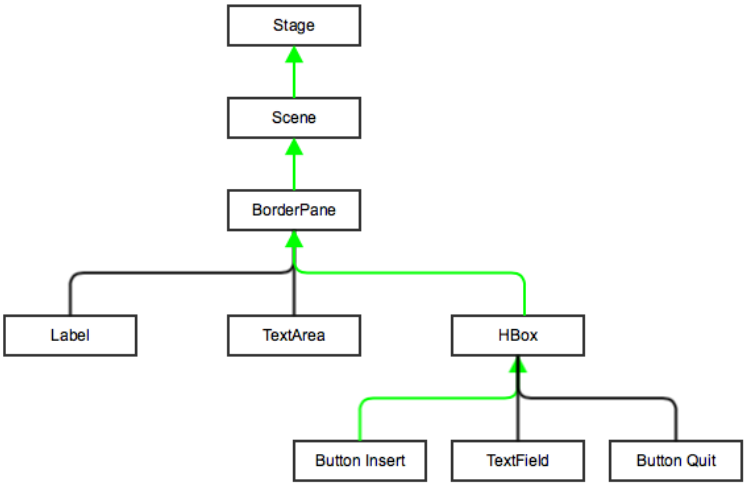


FIGURE 18 – Illustration de l'`event dispatch chain` en phase montante

Le traitement des événements comporte les étapes suivantes :

- La sélection de la cible de l'événement. Si plusieurs composants se trouvent à un emplacement donné c'est celui qui est « au-dessus » qui est considéré comme la cible ;
- La détermination de la chaîne de traitement des événements (*Event Dispatch Chain*). Ce chemin part de la racine (*Stage*) et va jusqu'au composant cible en parcourant tous les nœuds intermédiaires ;
- Le traitement des filtres d'événement (*Event Filter*). Exécute le code des filtres en suivant le chemin descendant, de la racine (*Stage*) jusqu'au composant cible ;
- Le traitement des gestionnaires d'événement (*Event Handler*). Exécute le code des gestionnaires d'événement en suivant le chemin montant, du composant cible à la racine (*Stage*) ;
- Si un filtre ou un gestionnaire d'événement consomme l'événement, celui-ci ne sera pas propagé au nœud suivant de la chaîne de traitement.

Si on souhaite que le bouton `Insert` de la classe `PrintText` imprime le texte encodé à l'écran, plusieurs solutions sont possibles, tel que présenté dans les sections suivantes.

5.1 Gestionnaire d'événement : le handler

Afin de créer le handler qui va appeler la méthode à exécuter lorsque l'événement est déclenché, l'une des solutions possibles est de créer une classe. Dans le cas présent, on l'intitule `InsertButtonHandler`.

Cette classe implémente l'interface `EventHandler` et effectue les opérations souhaitées via la méthode `handle()`.

```
1  public class InsertButtonHandler
2          implements EventHandler<ActionEvent> {
3
4      private final TextArea tArea;
5      private final TextField tfdText;
6      //—— Constructeur ————————————————————————
7      public InsertButtonHandler(TextArea tArea, TextField tfdText) {
8          this.tArea = tArea;
9          this.tfdText = tfdText;
10     }
11
12     //—— Code exécuté lorsque l'événement survient ——
13     @Override
14     public void handle(ActionEvent event) {
15         tArea.appendText(tfdText.getText());
16     }
17 }
```

Remarquez que le constructeur du gestionnaire d'événement reçoit en paramètres les composants impliqués lors du traitement de l'événement. Dans le cas présent, il s'agit des composants `TextArea` et `TextField`.

Au sein de la classe `PrintText` il faut créer une instance de `InsertButtonHandler` et l'enregistrer comme gestionnaire d'événement sur le bouton `Insert` en invoquant la méthode `addEventHandler()`.

```
1          //—— Button Events Handling
2          InsertButtonHandler insertCtrl = new InsertButtonHandler(txaMsg,tfdCharacter);
3          btnInsert.addEventHandler(ActionEvent.ACTION, insertCtrl);
```

La méthode `addEventHandler` [18] agit sur une instance de `Node` et prend en paramètres le type d'événement auquel il doit réagir ainsi que le gestionnaire de cet événement.

5.2 Gestionnaire d'événement : classes imbriquées et classes anonymes

Jusqu'à présent, les classes utilisées dans ce document étaient toutes des classes de premier niveau, c'est-à-dire des membres directs des paquetages, sans imbrication.

Parfois, ce choix de design est fastidieux, surtout si les responsabilités de ces classes sont limitées. En particulier, dans l'exemple actuel, on remarque que la seule fonctionnalité de la classe `InsertButtonHandler` est d'exécuter l'unique instruction `tArea.appendText(tfdText.getText())`.

Le langage *Java* permet cependant de définir des classes à l'intérieur d'autres classes. Il s'agit de classes imbriquées ou de *Nested Class*. Elles sont classées en deux catégories :

Les deux types de classes imbriquées :

- Les classes imbriquées statiques (*Static Nested Class*)
- Les classes imbriquées non statiques (*Inner Class*)
  - Les classes locales (*Local Inner Class*)
  - Les classes anonymes (*Anonymous Inner Class*)

Dans le cadre de ce document, on utilisera uniquement les classes anonymes. L'étudiant intéressé trouvera plus d'informations à ce sujet dans le tutoriel Oracle [19]. Il est toutefois recommandé de consulter le tutoriel concernant l'ensemble des types de classes internes [20].

Ainsi, comme le handler pour le bouton `Insert` est uniquement utilisé pour ce composant précis, il semble avisé de se passer de la création de la classe `InsertButtonHandler` et d'utiliser une classe sans nom, dite anonyme.

```
1        //--- Button Events Handling
2        btnInsert.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
3
4            @Override
5            public void handle(ActionEvent event) {
6                txaMsg.appendText(tfdCharacter.getText());
7            }
8        });
```

5.3 Gestionnaire d'événement : méthode `setOn`*`EventType`*

La plupart des composants possèdent des méthodes dédiées à l'enregistrement d'un gestionnaire particulier. Par exemple, comme le montre l'exemple ci-dessous, pour enregistrer un gestionnaire sur le bouton `Insert` il faut appeler la méthode `setOn`*`Action`*`()` et lui passer en paramètre une instance de classe anonyme qui implémente l'interface `EventHandler`.

```
1        //--- Button Events Handling
2        btnInsert.setOnAction(new EventHandler<ActionEvent>() {
3
4            @Override
```

18. https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/Node.html
19. https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html
20. https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

```
5               public void handle(ActionEvent event) {
6                   txaMsg.appendText(tfdCharacter.getText());
7               }
8           });
```

Pour chaque composant utilisé, il faut faire l'inventaire des méthodes `setOn`*`EventType`* disponibles. À ce titre, la table 2 illustre une liste des actions associées aux méthodes `setOn`*`EventType`* les plus courantes.

Composant	Evènement	Méthode
`Node`	`KeyEvent`	Pression sur une touche du clavier
`Node`	`MouseEvent`	Déplacement de la souris ou pression sur un de ses boutons
`Node`	`MouseDragEvent`	Glisser-déposer avec la souris
`Node`	`ScrollEvent`	Composant scrollé
`ButtonBase`	`ActionEvent`	Bouton cliqué
`ComboBoxBase`	`ActionEvent`	`ComboBox` ouverte ou fermée
`ContextMenu`	`ActionEvent`	Une des options d'un menu contextuel activée
`MenuItem`	`ActionEvent`	Option de menu activée
`TextField`	`ActionEvent`	Pression sur Enter dans un champ texte
`Menu`	`Event`	Menu est affiché ou masqué
`Window`	`WindowEvent`	Fenêtre affichée, fermée, masquée

TABLE 2 – Liste des actions associées aux méthodes `setOn`*`EventType`*

5.4 La méthode `consume()`

Pour rappel, avant d'arriver jusqu'aux gestionnaires, un événement doit passer au travers des différents filtres présents dans la chaîne de traitement. Ceux-ci peuvent avoir comme effet de consommer l'événement et de l'empêcher de passer au nœud suivant.

Cette « destruction » de l'événement en question est réalisée par la méthode `consume` des filtres.

5.5 Multiplicité des filtres et des gestionnaires

Si un nœud du graphe de scène possède plusieurs récepteurs d'événements enregistrés, l'ordre d'activation de ces récepteurs sera basé sur la hiérarchie des types d'événement.

— Un récepteur pour un type spécifique sera toujours exécuté avant un récepteur pour un type plus générique. Par exemple, un filtre enregistré pour `MouseEvent.MOUSE_PRESSED` sera exécuté avant un filtre pour `MouseEvent.ANY` qui sera exécuté avant un filtre pour `InputEvent.ANY`. Pour rappel, la hiérarchie des types d'événements est illustrée à la figure 16, page 16.

— L'ordre d'exécution des récepteurs pour des types de même niveau n'est pas défini.

6 Design d'un composant simple

Les librairies graphiques telles que Swing ou JavaFX offrent une grande variété de composants réutilisables qui permettent de développer rapidement les interfaces utilisateurs. On trouvera notamment différentes sortes de boutons, des menus, des éléments pour afficher ou introduire du texte, des listes ou des images, ou encore des composants plus complexes comme des éditeurs HTML ou des graphes.

Malgré la richesse des composants disponibles, il est toutefois fréquent d'avoir besoin de composants ad hoc.

En général, et en particulier pour JavaFX, un composant graphique doit satisfaire plusieurs propriétés fondamentales :

— Il doit être intégrable au graphe de scène. En JavaFX, cela impose au composant d'être une sous-classe (directe ou non) de la classe `Node`.
— Il expose ses propriétés au travers d'accesseurs et mutateurs (`get` et `set`).
— Il dispose d'un mécanisme de notification de ses changements d'état (les observateurs enregistrés seront avertis).

En JavaFX, il existe différentes manières de créer un nouveau composant. Certaines d'entre elles sont listées ci-dessous.

1. Étendre la classe `Parent` : permet d'ajouter des enfants pour construire le composant, mais ne gère pas les `css`.
2. Étendre la classe `Region` et ajouter du style avec un `css`.
3. Étendre une classe conteneur telle une `VBox`, une `HBox`, ou encore une `Pane`.
4. Définir un composant à l'aide d'un fichier `FXML`.
5. Étendre la classe `Canvas` ou `Region` et utiliser un `Canvas` par composition pour l'affichage. Un `Canvas` n'a pas de noeuds enfants et il n'est pas possible d'avoir des interactions sur des parties du canvas (seulement sur l'entièreté). Un `Canvas` pourrait être nécessaire si le composant est un dessin complexe.
6. Une classe `Control` + classe `Skin` + `css` : cette approche est l'approche standard pour des composants complexes avec une interaction riche (calendrier, menu déroulant, etc). Elle ne sera pas abordé dans ce document.

Une option très souple pour des rendus plus complexes consiste à coupler l'une ou l'autre approche ci-dessus et de passer par la classe `SVGPath` et un fichier SVG (Scalable Vector Graphics [21]) pour l'affichage. Un fichier SVG peut-être créé à l'aide d'un des nombreux éditeurs existants, par exemple InkScape [22].

Vous trouverez en ligne beaucoup d'informations et de discussions sur les différentes manières de créer des composants, par exemple :

— 'Benjamin's programming Blog' [23],
— 'Adding a Custom JavaFX Component to Scene Builder 2.0' [24],
— 'Be Creative and Create Your Own JavaFX 8 Controls' [25].

21. https://fr.wikipedia.org/wiki/Scalable_Vector_Graphics
22. https://inkscape.org/en/
23. https://programmingwithpassion.wordpress.com/2013/07/07/creating-a-reusable-javafx-custom-control/
24. https://rterp.wordpress.com/2014/07/28/adding-custom-javafx-component-to-scene-builder-2-0-part-2/
25. https://www.youtube.com/watch?v=7PPcM0E5yQw

6.1 Les propriétés JavaFX

Pour JavaFX, une propriété est un élément d'une classe que l'on peut manipuler à l'aide de getters et de setters. En plus des getter et setter habituels, les propriétés JavaFX possèdent une méthode qui retourne un objet implémentant l'interface Property [26].

Cette interface, l'interface ReadOnlyProperty [27] et de nombreuses propriétés JavaFX font partie du package javafx.beans.property [28]. Les classes de ce package, préfixées par *ReadOnly*, ne permettront que la lecture de la valeur de la propriété.

L'intérêt de ces propriétés est de pouvoir être liées entre-elles (*Binding*). Autrement dit, le changement d'une propriété entraîne automatiquement la mise à jour d'une autre. Elles peuvent également déclencher un événement lorsque leur valeur change et provoquer la réaction adaptée par le gestionnaire d'événement.

Pour illustrer les différentes notions liées aux propriétés, plusieurs implémentations d'un composant sont proposées [29]. Il s'agit d'un LED [30].

Ce composant possède deux propriétés :
— `on` de type booléen,
— `color` de type `Color`.

Notez les différences de ces implémentations comme la classe héritée, le composant JavaFX contenu, la taille fixe ou variable du composant.

Ainsi, le composant ci-dessous est une sous-classe de `Parent` et utilise un `Circle` pour se dessiner. Sa taille est fixe.

```
1  public class Led extends Parent {
2      private final BooleanProperty on;
3      private final ObjectProperty<Color> color;
4      private final Circle circle;
5
6      public Led() {
7    on = new SimpleBooleanProperty(true);
8    color = new ObjectPropertyBase<Color>() {
9                  @Override
10                 public Object getBean() {
11                     return this;
12                 }
13                 @Override
14                 public String getName() {
15                     return "Color";
16                 }
17             };
18         circle = new Circle(50);
19         circle.setStroke(Color.BLACK);
20         getChildren().add(circle);
21         setColor(Color.RED);
22     }
23     public final void setOn(boolean on) {
24         this.on.set(on);
25         circle.setFill(on?color.get():Color.TRANSPARENT);
26     }
27     public final boolean isOn() {
28         return on.get();
29     }
30     public final BooleanProperty onProperty() {
```

26. https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/Property.html
27. https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/ReadOnlyProperty.html
28. https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/package-summary.html
29. Ces implémentations se trouvent dans `LedPF.java`, `Led.java` et `canvas.java`. En l'état, ces implémentations ne compilent pas, il faut transférer le code au sein d'un fichier appelé `Led.java`, définissant une classe `Led`
30. LED : light-emitting diode ou, en français, diode électroluminescente

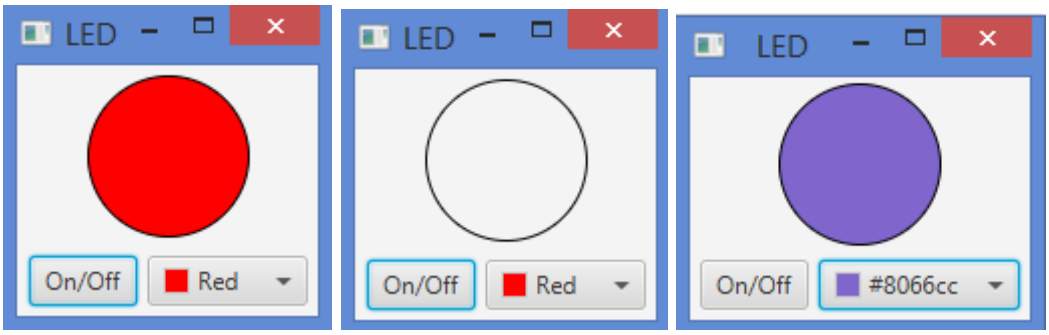
```
31          return on;
32      }
33      public final Color getColor() {
34          return color.get();
35      }
36      public final void setColor(Color color) {
37          this.color.set(color);
38          circle.setFill(isOn()?getColor():Color.TRANSPARENT);
39      }
40      public final ObjectProperty<Color> colorProperty() {
41          return color;
42      }
43  }
```

Les propriétés `on` et `color` du composant LED sont des instances des classes `SimpleBooleanProperty`[31] et `ObjectPropertyBase<Color>`[32]. Remarquez pour cette dernière, l'implémentation des méthodes `getBean` et `getName` par une classe anonyme.

Les propriétés JavaFX permettent de représenter une valeur d'un type primitif, d'une chaîne de caractères, de certaines collections à l'aide d'une classe prédéfinie spécifique à ce type (comme `BooleanProperty`[33] pour le type booléen de `on`) ou à l'aide d'une classe pour le type `Objet` (comme `ObjectProperty<Color>`[34] pour le type `Color`).

Vous pouvez tester les mutateurs des propriétés du composant en créant une application qui
- change la valeur de la propriété `On` en cliquant sur le bouton *On/Off*,
- change la valeur de la propriété `Color` en choisissant une couleur dans le sélecteur de couleurs.

Les différents effets de ces changements de propriétés sont illustrés à la figure 19.



(a) (b) (c)

FIGURE 19 – Effet des changements de propriétés

6.2 Les propriétés sont observables

Toute propriété JavaFX, en plus des méthodes pour consulter et modifier la valeur, offre aussi des méthodes pour observer les changements. En effet, la possibilité d'enregistrer des observateurs (*Listener)* qui seront avertis lorsque la valeur de la propriété change ou quand une invalidation

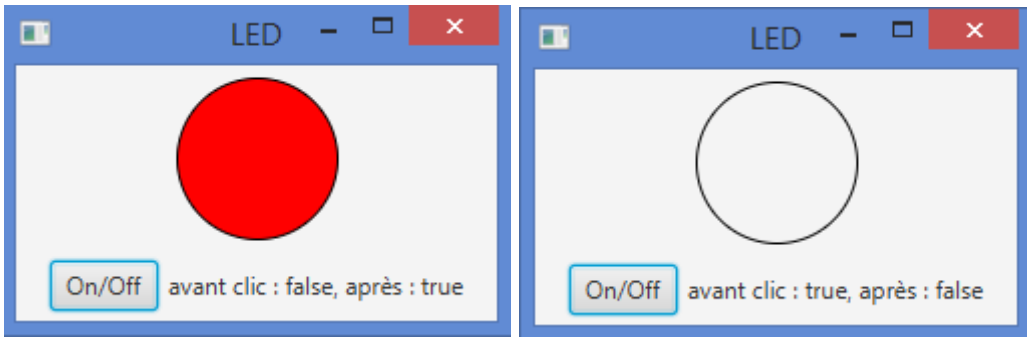
31. https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/SimpleBooleanProperty.html
32. https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/ObjectPropertyBase.html
33. https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/BooleanProperty.html
34. https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/ObjectProperty.html

survient est offerte par l'implémentation des interfaces *ObservableValue* et *Observable* par toute classe de propriété *JavaFX*.

6.2.1 Les interfaces *ObservableValue* et *ChangeListener*

Pour illustrer l'écoute par un `ChangeListener` du changement de valeur d'une propriété, on peut remplacer le sélecteur de couleurs dans l'application de test par un label `onOffTxt` et ajoutez l'écoute du changement de valeur de la propriété `on` avec le code suivant.

```
1        final Label onOffTxt = new Label();
2        led.onProperty().addListener(
3                (ObservableValue<? extends Boolean> observable,
4                        Boolean oldValue, Boolean newValue) -> {
5                    onOffTxt.setText("avant clic : " + oldValue + ", après : " + newValue);
6                });
7        box.getChildren().add(onOffTxt);
```



(a) (b)

FIGURE 20 – Écoute du changement de la propriété `on`

Lorsque le bouton est cliqué, la propriété `on` est modifiée et cette modification est notifiée au *listener* qui adapte le texte du label avec l'ancienne valeur suivie de la nouvelle.

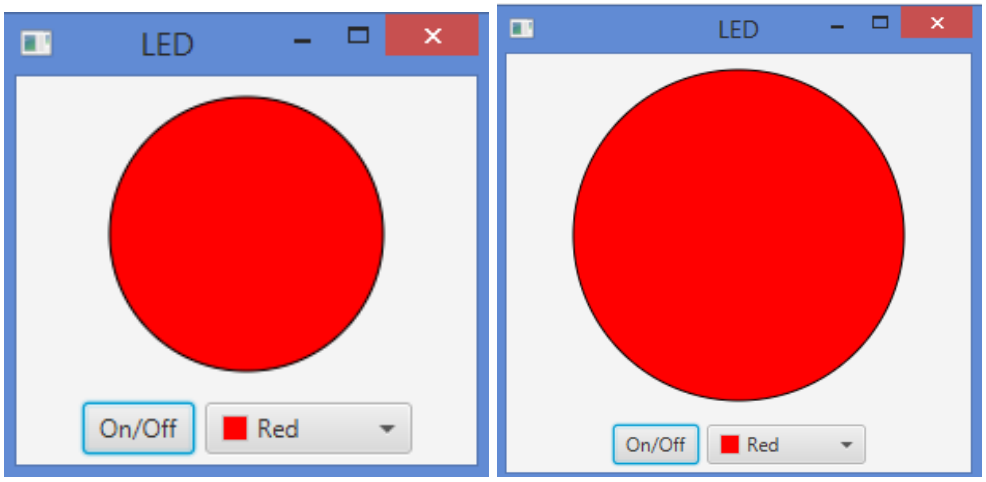
6.2.2 Les interfaces *Observable* et *InvalidationListener*

Pour illustrer l'écoute par un `InvalidationListener` de l'invalidation de valeur d'une propriété, vous disposez du composant `Led` (implémenté dans `canvas.java`). Cet exemple permet également de montrer une autre manière de dessiner le led : en utilisant un Canvas à la place d'un Circle.

```
1  public class Led extends Region {
2      private static final double  BORDER_RSIZE = 0.02;
3      private double size;
4    ...
5      public Led() {
6      ...
7          addListeners();
8      }
9      private void addListeners() {
10         widthProperty().addListener(o -> resizeCanvas());
11         heightProperty().addListener(o -> resizeCanvas());
12     }
13     public final void setOn(boolean on) {
14         this.on.set(on);
15         repaint();
16     }
```

```
17    ...
18      public final void setColor(Color color) {
19          this.color.set(color);
20          repaint();
21      }
22    ...
23      private void resizeCanvas() {
24          double width = getWidth();
25          double height = getHeight();
26
27          // resizing the canvas to the biggest possible square in the parent (region)
28          size = width < height ? width : height;
29          canvas.setWidth(size);
30          canvas.setHeight(size);
31
32          // centering the canvas in the parent (region)
33          if (width > height) {
34              canvas.relocate((width - size)*0.5, 0);
35          } else if (height > width) {
36              canvas.relocate(0, (height - size)*0.5);
37          }
38          repaint();
39      }
40      private void repaint() {
41          double ulc = BORDER_RSIZE * size;          // canvas's upper left corner
42          double brc = (1-2*BORDER_RSIZE) * size;    // canvas's bottom right corner
43
44          context.clearRect(0, 0, size, size);         // clean the canvas
45          context.strokeOval(ulc, ulc, brc, brc);      // draw the led's border
46
47          if (isOn()) {       // fill the led with the color property value if is is on
48              context.setFill(color.get());
49              context.fillOval(ulc, ulc, brc, brc);
50          }
51      }
52  }
```

Dans `addListeners`, un `InvalidationListener` est ajouté aux propriétés largeur et hauteur de la `Region` qui est parent de `Canvas`. Lorsqu'une des propriétés est invalidée, l'implémentation de la méthode `invalidated` de l'interface fonctionnelle `InvalidationListener` consiste à appeler la méthode `resizeCanvas` qui calcule la taille idéale du `Canvas` et redessine le led en appelant la méthode `repaint`.



(a) (b)

6.3 Le binding

Il est possible de lier des propriétés JavaFX afin de modifier automatiquement la valeur d'une propriété en fonction d'une autre. Comme, par exemple, modifier la valeur d'un *label* lorsque le

curseur d'un *slider* est déplacé. Il s'agit du mécanisme de *binding*.

6.3.1 Binding unidirectionnel ou bidirectionnel

Le binding peut être unidirectionnel ou bidirectionnel :
— unidirectionnel : la modification d'une propriété `p1` entraînera la modification d'une autre propriété `p2` mais pas l'inverse. Lorsque la valeur de la propriété `p2` dépend de la valeur de la propriété `p1`, la méthode `bind` permet de créer ce lien : `p2.bind(p1)`. La méthode `unbind` permet de le supprimer.
— bidirectionnel : la modification d'une propriété `p1` entraînera la modification d'une autre propriété `p2` et inversément. La méthode `bindBidirectional()` permet d'établir une liaison bidirectionnelle : `p2.bindBidirectional(p1)` ou `p1.bindBidirectional(p2)`. La méthode `unbindBidirectional` permet de supprimer un lien bidirectionnel.

6.3.2 Propriétés calculées

Il est également possible de créer des propriétés calculées :
— en utilisant les méthodes statiques de la classe `Bindings` pour effectuer des opérations,
— en utilisant les méthodes, qui peuvent être chainées, disponibles dans les classes des propriétés,
— en combinant les deux.

L'implémetation de `Led` dans `Led.java` est un variante des précédentes : c'est une sous-classe de `Region` et du binding a été ajouté pour que le centre et le rayon du cercle qui représente le LED soient automatiquement modifiés lorsque la taille de la `Region` est modifiée.

```
1     private void doBindings() {
2         circle.centerXProperty().bind(widthProperty().divide(2));
3         circle.centerYProperty().bind(heightProperty().divide(2));
4         circle.radiusProperty().bind(
5                 Bindings.subtract(Bindings.min(widthProperty().divide(2),
6                                                heightProperty().divide(2)),
7                                   10));
8     }
```

6.3.3 *Binding* de bas niveau

Un binding de bas niveau (*low-level binding*) peut également être réalisé en implémentant la méthode abstraite `computeValue()` d'une des classes de *binding* (`BooleanBinding`, `ObjectBinding`, etc.). Le composant `esi.atl.component.rvlb.Led` est une variante de `esi.atl.component.rv.Led` avec un *binding* de bas niveau pour la couleur du cercle.

```
1  // ajout de circle.fillProperty().bind(new FillShapeBinding(color, on)); dans doBindings()
2
3  public class FillShapeBinding extends ObjectBinding<Color> {
4
5      private final ObjectProperty<Color> color;
6      private final BooleanProperty on;
7
8      public FillShapeBinding(final ObjectProperty<Color> color, final BooleanProperty on) {
9          super();
10         this.color = color;
11         this.on = on;
12         bind(color, on);
```

```
13      }
14
15      @Override
16      protected Color computeValue() {
17          return on.get()?color.get():Color.TRANSPARENT;
18      }
19
20  }
```