Interfaces graphiques avec JavaFX

Michel Schinz

2017-05-08

1 Introduction

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

- **AWT** (*Abstract Window Toolkit*), tombé en désuétude mais servant de base à Swing,
- **Swing**, utilisé pour la plupart des applications jusqu'à récemment, mais en cours de remplacement par JavaFX, et
- **JavaFX**.

Cette leçon présente une vue d'ensemble des principaux concepts de JavaFX, la plupart d'entre eux se retrouvant, sous une forme ou une autre, dans des bibliothèques similaires pour d'autres langages de programmation.

Au vu de la taille de la bibliothèque JavaFX, il est impossible ici de la décrire de manière détaillée, et certains aspects importants ont même été totalement omis. Pour plus d'information, il est donc recommandé de se reporter aux références données au bas de ce document.

2 Concepts fondamentaux

Une interface graphique JavaFX est constituée d'un certain nombre de **nœuds** (*nodes*), qui peuvent être de différente nature : simples formes géométriques (cercles, lignes, etc.), composants d'interface utilisateur (bouton, menu, etc.), conteneurs, etc.

Ces nœuds sont organisés en une hiérarchie que l'on nomme le **graphe de scène** (*scene graph*). Malgré son nom, le graphe de scène est un simple arbre, c'est-à-dire un graphe acyclique dans lequel chaque nœud a au plus un parent. Cet arbre reflète l'organisation des nœuds à l'écran, dans la mesure où un descendant d'un nœud dans le graphe de scène apparaît généralement à l'écran imbriqué dans son ancêtre.

La figure ci-dessous présente un exemple très simple d'un graphe de scène et la manière dont les nœuds qui le composent apparaissent à l'écran, imbriqués conformément à la hiérarchie. Les noms des nœuds du graphe de scène sont ceux des classes JavaFX correspondant à ce type de nœud.

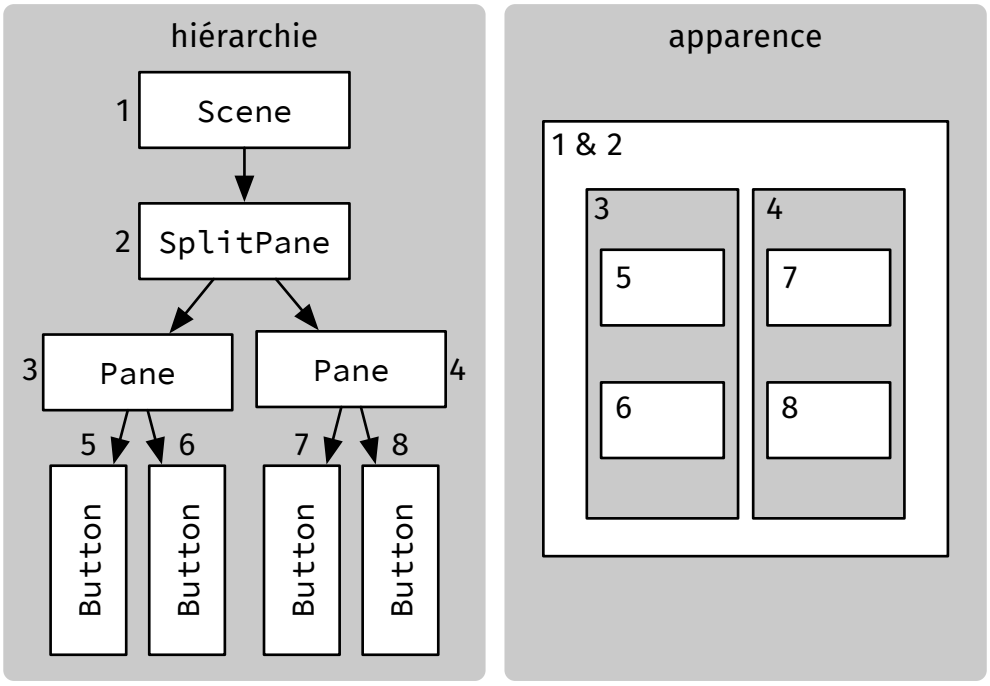

Fig. 1: Graphe de scène et son apparence à l'écran

Comme cet exemple l'illustre, on peut distinguer plusieurs genres de nœuds JavaFX qui, des feuilles à la racine de l'arbre, sont :

- les nœuds de base, qui apparaissent aux feuilles de la hiérarchie et n'ont donc aucun descendant ; ils représentent généralement des entités atomiques simples comme des figures géométriques ou des boutons,
- les nœuds intermédiaires (autres que la racine), qui possèdent un certain nombre de nœuds enfants et dont le but est généralement d'organiser ces enfants à l'écran, p.ex. en les plaçant côte à côte ; beaucoup d'entre eux sont ce que JavaFX nomme des panneaux,
- la racine, qui est le seul nœud du graphe de scène dénué de parent, qui doit toujours avoir le type `Scene` et dont la représentation graphique est presque inexistante.

Pour que le contenu d'un graphe de scène apparaisse effectivement à l'écran, l'objet `Scene` à sa racine doit être placé dans un objet de type `Stage`. Ce type représente un conteneur graphique de base du système sur lequel le programme s'exécute. Par exemple,

sur un ordinateur de bureau, le type `Stage` représente généralement une fenêtre du système d'exploitation[1].

L'exemple ci-dessous illustre la création d'un graphe de scène très simple correspondant à un formulaire de connexion.

```
Label nameL = new Label("Nom␣:");
TextField nameF = new TextField();

Label pwL = new Label("Mot␣de␣passe␣:");
TextField pwF = new TextField();

Button connectB = new Button("Connexion");

GridPane grid = new GridPane();
grid.addRow(0, nameL, nameF);
grid.addRow(1, pwL, pwF);
grid.add(connectB, 0, 2, 2, 1);

GridPane.setHalignment(connectB, HPos.CENTER);

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

Le graphe de scène créé par cet exemple est présenté à la figure 2, et une copie d'écran du résultat est présenté à la figure 3.

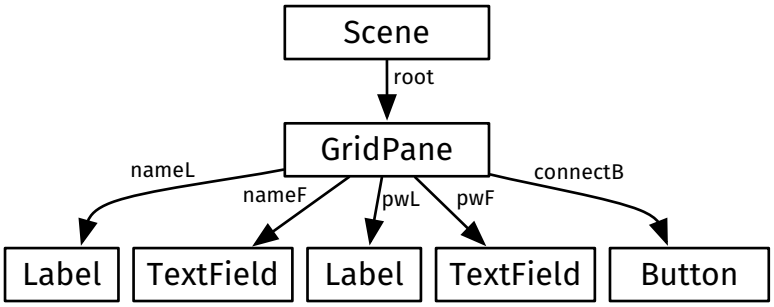

Fig. 2: Graphe de scène simple

1. Malheureusement, les mots anglais *scene* et *stage* se traduisent tous les deux par le mot *scène* qui a deux significations en français : une scène est à la fois une partie d'un spectacle (*scene* en anglais) et le lieu sur lequel les artistes se produisent (*stage* en anglais). Pour cette raison, aucune tentative de traduction de ces termes n'est faite ici.

Fig. 3: Interface correspondant au graphe de scène simple

3 Nœuds

Tous les types de nœuds JavaFX héritent d'une classe-mère commune nommée `Node`. Cette classe possède un très grand nombre d'attributs permettant, entre autres, de :

- positionner le nœud dans le système de coordonnées de son parent, par exemple en le translatant ou en le tournant,
- définir le style visuel du nœud, par exemple son opacité,
- attacher des gestionnaires d'événements, décrits à la section 5.1, qui sont informés lorsque différents événements liés au nœud se produisent, par exemple son survol par le pointeur de la souris.

Les principales catégories de nœuds JavaFX sont décrites ci-après.

3.1 Nœuds géométriques

Les nœuds géométriques représentent des formes géométriques diverses ou du texte. Tous héritent d'une classe-mère commune nommée `Shape`, héritant elle-même de `Node`. Les principaux types de nœuds graphiques représentent :

- des formes géométriques simples (classes `Arc`, `Circle`, `Ellipse`, `Line`, `Polygon`, `Polyline` et `Rectangle`),
- des courbes de Bézier (`CubicCurve` et `QuadCurve`),
- des « chemins », composés d'une succession d'éléments simples comme des segments de lignes ou de courbes, etc. (`Path` et `SVGPath`),
- du texte, qui peut occuper plusieurs lignes (`Text`).

3.2 Nœuds graphiques

Les nœuds graphiques présentent un contenu graphique, qui peut être une image ou une vidéo. Ces nœuds, qui héritent directement de la classe `Node`, sont :

- `ImageView`, qui affiche une image,
- `MediaView`, qui affiche une vidéo,
- `Canvas`, qui affiche une image modifiable, sur laquelle il est possible de dessiner différentes primitives graphiques (lignes, rectangles, etc.).

La différence entre les primitives dessinées sur un canevas et les nœuds géométriques de la section précédente est que ces derniers ont une existence propre dans le graphe de scène et peuvent être manipulés individuellement, ce qui n'est pas le cas des premières. Par exemple, un nœud de type `Line` appartenant à un graphe de scène peut être déplacé ou tourné, alors qu'une ligne dessinée sur un canevas ne peut plus être manipulée une fois dessinée.

Cette différence est semblable à celle existant entre les programmes de dessin vectoriels — qui représentent un dessin au moyen d'objets similaires aux nœuds géométriques — et les programmes de dessin *bitmap* — qui représentent un dessin comme un canevas modifiable.

3.3 Nœuds de contrôle

Les nœuds de contrôle sont des nœuds représentant des composants d'interface graphique avec lesquels il est possible d'interagir, comme p.ex. les boutons, les menus, etc. Tous héritent d'une classe-mère commune nommée `Control`, héritant (indirectement) de `Node`.

Le terme *contrôle* rappelle le fait que ces nœuds font partie du contrôleur de l'application, selon la terminologie du patron MVC. Cela dit, la quasi totalité d'entre eux font également partie de la vue, puisqu'ils présentent graphiquement une valeur.

3.3.1 Informations

Certains nœuds de contrôle sont en réalité passifs et ont pour unique but de présenter une information, sans permettre sa modification par l'utilisateur.

La classe `Label` représente une étiquette, c-à-d un court texte qui permet de clarifier l'utilité de certaines parties de l'interface graphique. Par exemple, à chaque champ textuel d'une interface est généralement associée une telle étiquette décrivant le contenu du champ.

Les classes `ProgressIndicator` et `ProgressBar` permettent de visualiser de différentes manières l'état d'avancement d'une opération de longue durée, p.ex. le téléchargement de données, la progression d'un calcul, etc.

3.3.2 Boutons

Les nœuds `Button`, `ToggleButton` et `RadioButton` représentent différents types de boutons sur lesquels il est possible d'appuyer, généralement en cliquant au moyen de la souris.

La classe `Button` représente un bouton à un état, sur lequel il est possible d'appuyer dans le but d'effectuer une action. Par exemple, dans un navigateur Web, il existe généralement un bouton à un état permettant de recharger la page courante.

Les classes `ToggleButton` et `RadioButton` représentent des boutons à deux états, c-à-d qui peuvent être soit activés, soit désactivés.

L'état d'un bouton de type `ToggleButton` peut être inversé (c-à-d passé d'activé à désactivé, ou vice versa) par un clic de la souris. Ce type de bouton est souvent utilisé dans les réglages des applications pour activer ou désactiver une option donnée.

Un bouton de type `RadioButton` fait toujours partie d'un groupe de boutons du même type, et exactement un d'entre eux peut être sélectionné à un moment donné. Dès lors, l'activation d'un bouton radio par un clic de la souris provoque la désactivation de (l'unique) autre bouton activé dans le même groupe. Le nom vient du fait que ce type de bouton était fréquent sur les récepteurs radio.

3.3.3 Editeurs de valeurs atomiques

Un certain nombre de nœuds de contrôle permettent de présenter et éventuellement modifier des valeurs atomiques, c-à-d indivisibles.

Les classes `HTMLEditor`, `TextArea` et `TextField` permettent de présenter et éditer du texte. La première gère du texte « stylé », p.ex. en gras, italique et autre. La seconde gère du texte non stylé mais pouvant faire plusieurs lignes. La dernière gère une seule ligne de texte non stylé. A noter que `TextField` permet également de présenter et éditer une valeur quelconque qui peut être représentée de manière textuelle, p.ex. un nombre.

La classe `Slider` permet de présenter et modifier une valeur, souvent continue, comprise entre deux bornes.

La classe `Spinner` permet de présenter et modifier une valeur de manière similaire à un champ textuel de type `TextField`, à condition que la valeur fasse partie d'un ensemble ordonné, p.ex. les entiers. Des petits boutons attachés au champ permettent de passer de la valeur actuelle à la précédente ou à la suivante dans l'ordre.

La classe `ChoiceBox` permet de présenter et d'effectuer un choix parmi un certain nombre (fini) de valeurs. La valeur actuellement choisie est affichée, et un menu contenant la totalité des choix possible est présenté lors d'un clic sur le nœud.

La classe `ColorPicker` permet de présenter et choisir une couleur, tandis que la classe `DatePicker` permet de présenter et choisir une date.

3.3.4 Editeurs de valeurs composites

Un certain nombre de nœuds de contrôle permettent de présenter et éventuellement modifier des valeurs composites, c-à-d constituées d'un nombre variable d'autres valeurs.

La classe `ListView` présente une liste de valeurs et offre la possibilité à l'utilisateur de sélectionner une ou plusieurs d'entre elles.

La classe `TableView` présente une table de valeurs, c-à-d une liste de valeurs dotées de plusieurs attributs occupant chacun une colonne. Tout comme `ListView`, ce nœud gère également la sélection d'une ou plusieurs valeurs de la liste. De plus, il permet le tri des éléments de la table en fonction de l'une ou l'autre des colonnes.

La classe `TreeView` présente un arbre de valeurs, c-à-d des valeurs organisées en hiérarchie, et gère également la sélection d'une ou plusieurs d'entre elles.

3.4 Panneaux

Un panneau (*pane*) est un nœud dont le but principal est de servir de conteneur à ses nœuds enfants et, souvent, de les organiser spatialement d'une manière ou d'une autre.

Le type le plus simple de panneau est représenté par la classe `Pane`. Contrairement à la plupart des autres panneaux, celui-ci n'organise pas spatialement ses enfants. Il est donc nécessaire de les placer « manuellement », ce qui offre une flexibilité maximale, au prix bien entendu d'un travail supplémentaire.

Les autres types de panneaux sont représentés par des sous-classes de `Pane`, qui se distinguent par la manière dont ils organisent leurs enfants. A noter que, lors de l'organisation de ses enfants, un panneau peut être amené à redimensionner ceux qui sont redimensionnables. Cela implique que les nœuds redimensionnables aient un moyen de communiquer à leur parent des informations sur leur taille. Dans ce but, la classe `Node` est équipée d'un certain nombre de méthodes, permettant entre autres au parent de connaître :

- la largeur/hauteur minimale (`minWidth` / `minHeight`),
- la largeur/hauteur maximale (`maxWidth` / `maxHeight`), et
- la largeur/hauteur préférée, c-à-d idéale, (`prefWidth` / `prefHeight`)

de chacun de ses enfants. Les nœuds dont la taille est fixe ont la même valeur minimale, maximale et préférée pour ces deux dimensions, ce qui n'est bien entendu pas le cas des nœuds redimensionnables.

3.4.1 `BorderPane`

Un panneau de type `BorderPane` est doté de cinq zones : une zone centrale et quatre zones latérales (haut, bas, gauche et droite). Chaque zone peut être occupée par au plus un nœud, et toutes les zones sauf la centrale sont dimensionnées en fonction du nœud qu'elles contiennent. La totalité de l'espace restant est attribué à la zone centrale.

3.4.2 GridPane

Un panneau de type GridPane organise ses enfants dans une grille. La largeur de chaque colonne et la hauteur de chaque ligne sont déterminées en fonction de la taille des nœuds qu'elles contiennent.

Par défaut, les nœuds enfants n'occupent qu'une case de la grille, mais il est également possible de leur faire occuper une région rectangulaire composée de plusieurs lignes et/ou colonnes contiguës.

Finalement, les nœuds enfants peuvent être alignés de différentes manières (à gauche, à droite, au milieu, etc.) dans la case de la grille qu'ils occupent.

3.4.3 StackPane

Un panneau de type StackPane organise ses enfants de manière à les empiler les uns sur les autres, du premier (placé au bas de la pile) au dernier (placé au sommet).

Etant donné que ces enfants sont empilés, ils apparaissent visuellement les uns sur les autres. Tous sauf le premier sont donc en général partiellement transparents, faute de quoi ils obscurscissent totalement les nœuds placés sous eux.

3.4.4 ScrollPane

Un panneau de type ScrollPane donne accès à une sous-partie d'un nœud visuellement trop grand pour être affiché en totalité, et permet de déplacer la zone visualisée de différentes manières, p.ex. au moyen de barres de défilement.

3.4.5 TabPane

Un panneau de type TabPane est composé d'un certain nombre d'onglets (*tabs*) affichant chacun un nœud enfant différent. Seul un onglet est visible à un instant donné.

3.4.6 SplitPane

Un panneau de type SplitPane est divisé en plusieurs parties, chacune occupée par un nœud différent. La division peut être verticale ou horizontale, et est redimensionnable, éventuellement par l'utilisateur.

4 Propriétés et liens

Un problème récurrent dans les interfaces graphiques est celui de la synchronisation du modèle et de l'interface : comment faire pour que les valeurs affichées dans l'interface correspondent toujours à celles du modèle ?

Une solution assez naturelle, et souvent utilisée, consiste à rendre le modèle observable, au sens du patron *Observer*. Cela fait, les différents éléments de l'interface graphique peuvent s'enregistrer comme observateurs du ou des parties du modèle qu'ils représentent, et se mettre à jour lors d'un changement.

JavaFX offre un certain nombre de concepts et de classe facilitant la définition de modèles observables : les propriétés, les collections observables et les liens de données.

Avant de présenter ces concepts, il convient de noter une différence de terminologie entre la définition standard du patron *Observer* et les conventions de JavaFX : ce que le patron *Observer* appelle un sujet (*subject*) est appelé un observable (*observable*) ou une valeur observable (*observable value*) dans JavaFX ; de plus, ce que le patron *Observer* nomme un observateur (*observer*) est souvent appelé un auditeur (*listener*) dans JavaFX.

4.1 Propriétés

En JavaFX, une **propriété** (*property*) est une cellule observable, c-à-d un objet observable contenant une unique valeur. La valeur contenue par une propriété peut être soit un objet d'un type donné, soit une valeur d'un type primitif.

Une propriété peut être accessible en lecture seule (*read only*) ou en lecture et écriture (*read/write*). Bien entendu, les propriétés accessibles en lecture seule, par définition non modifiables, ne sont en général pas immuables, faute de quoi les observer n'aurait aucun intérêt.

JavaFX fournit plusieurs classes pour représenter les propriétés. Deux d'entre elles sont particulièrement utiles : `ObjectProperty`, classe abstraite et générique représentant une propriété d'un type objet donné, et `SimpleObjectProperty`, classe concrète qui lui correspond. L'extrait de programme ci-dessous montre comment créer une propriété contenant une chaîne, lui attacher un auditeur affichant sa nouvelle valeur chaque fois qu'elle est modifiée, et modifier sa valeur deux fois de suite.

```
ObjectProperty<String> p = new SimpleObjectProperty<>();
p.addListener((o, oV, nV) -> System.out.println(nV));

p.set(”hello”);                 // affiche ”hello”
p.set(”world”);                 // affiche ”world”
```

Les propriétés sont utiles, entre autres, pour stocker des valeurs simples qui apparaissent dans une interface graphique JavaFX.

Par exemple, une propriété contenant une chaîne de caractères se combine très bien avec un champ textuel permettant sa visualisation. Il suffit en effet au champ d'observer la propriété et de se mettre à jour à chacun de ses changements pour que la valeur affichée soit toujours celle du modèle.

De même, si un champ textuel stocke la chaîne qu'il affiche dans une propriété — ce qui est le cas — alors le modèle peut observer à son tour cette propriété et se mettre à jour dès qu'elle change. Cela garantit que les changements apportés à la valeur affichée dans le champ sont reflétées dans le modèle.

Comme nous le verrons plus bas, JavaFX offre un mécanisme de liaison permettant de mettre facilement en place cette synchronisation par observation mutuelle de deux propriétés.

4.2 Collections observables

Une propriété JavaFX contient une *unique* valeur observable. Or dans certaines situations, il est nécessaire de stocker une collection de valeurs dans un seul attribut observable.

Pour ce faire, JavaFX fournit, dans le paquetage `javafx.collections`, un ensemble de classes et interfaces représentant des collections observables. Ces collections sont, entre autres, utilisées par les composants capables de présenter et éventuellement modifier des valeurs composites, décrits à la section 3.3.4.

Les collections observables JavaFX sont calquées sur les collections Java « normales » (et non observables) du paquetage `java.util`. Ainsi, l'interface `ObservableList` de JavaFX étend l'interface `List` de la bibliothèque Java, en lui ajoutant principalement des méthodes liées à l'observation. De même, la classe `FXCollections` offre des méthodes quasi identiques à celles de la classe `Collections`, mais dont le comportement est préférable en présence de collections observables.

Par exemple, la méthode `sort` de `FXCollections`, qui trie une liste observable exactement comme la méthode `sort` de `Collections` trie une liste, s'arrange pour que les auditeurs de la liste ne soient averti du changement de son contenu qu'une fois le tri terminé. Ce comportement est préférable à celui de la méthode `sort` de `Collections`, dont chaque échange de deux éléments dans la liste à trier provoque une notification des auditeurs.

4.3 JavaFX *beans*

Plusieurs propriétés peuvent être combinées dans une classe afin de rendre chacun de ses attributs observables. Les classes de ce genre, dont tous les attributs sont des propriétés, sont très utiles à l'interface entre le modèle d'une application et son interface graphique.

JavaFX définit une convention pour la définition de ce type de classes, que l'on nomme pour des raisons historiques des *beans* JavaFX. Cette convention spécifie que :

- tout attribut de la classe doit être représenté par une propriété,
- la propriété correspondant à un attribut donné doit être accessible au moyen d'une méthode dont le nom est celui de la propriété suivi du suffixe `Property`,
- la valeur stockée dans la propriété doit également être accessible au moyen d'une méthode d'accès dont le nom est formé du préfixe `get` suivi du nom de la propriété (le préfixe `is` est également accepté pour les propriétés booléennes),

- si la propriété est accessible en lecture et écriture, une méthode de modification doit également être fournie, dont le nom est formé du préfixe `set` suivi du nom de la propriété.

Par exemple, un *bean* JavaFX `Person` représentant une personne et doté d'un attribut `firstName` accessible en lecture et écriture, et d'un attribut `age` accessible en lecture seule ressemble à :

```
public final class Person {
  public ObjectProperty<String> firstNameProperty() { … }
  public String getFirstName() { … }
  public void setFirstName(String newFirstName) { … }

  public ReadOnlyIntegerProperty ageProperty() { … }
  public int getAge() { … }
}
```

La quasi totalité des classes JavaFX, en particulier celles représentant les différents types de nœuds, sont des *beans* JavaFX. Dès lors, leurs attributs sont observables et peuvent être facilement liés aux attributs du modèle, ce qui est extrêmement utile, comme nous allons le voir.

4.4 Liens

On l'a vu, le fait que les valeurs du modèle de l'application soient observables est très utile pour la mise en œuvre de son interface graphique, car un composant graphique représentant une valeur du modèle peut l'observer et se mettre à jour en cas de changement.

Lors de l'écriture du code faisant le lien entre le modèle observable et l'interface graphique d'une application, il est donc très fréquent de devoir faire en sorte qu'un élément de l'interface graphique observe une valeur du modèle et se mette à jour lorsqu'elle change. Ce code est toujours le même, et il est donc intéressant de l'écrire une fois pour toutes et de le fournir dans une bibliothèque.

Pour cela, JavaFX offre la notion de **lien de données** (*data binding*), qui permet de garantir que la valeur d'une propriété est toujours égale à la valeur d'une autre propriété. Les liens peuvent être uni- ou bidirectionnels.

4.4.1 Liens unidirectionnels

Avec un lien unidirectionnel, une propriété donnée est observée et, dès qu'elle change, sa valeur est copiée dans une autre propriété.

Un lien unidirectionnel peut être établi entre deux propriétés au moyen de la méthode `bind`. Cette méthode s'applique à la propriété destination, c-à-d celle dont la valeur sera

toujours copiée de l'autre, et on lui passe la propriété source en argument. L'exemple ci-dessous illustre l'établissement d'un lien forçant la propriété `p1` à toujours avoir la même valeur que la propriété `p2`.

```
ObjectProperty<String> p1 = new SimpleObjectProperty<>();
ObjectProperty<String> p2 = new SimpleObjectProperty<>();

p1.addListener((o, oV, nV) -> System.out.println(nV));

p2.set("hello");                // n'affiche rien (!)
System.out.println("---␣binding␣p1␣to␣p2");
p1.bind(p2);                    // affiche "hello"
p2.set("world");                // affiche "world"
```

A noter qu'après l'établissement d'un tel lien, la modification directe de la propriété liée (`p1`) est interdit, et l'appel à sa méthode `set`, par exemple, provoque une exception.

Un lien unidirectionnel est utile p.ex. lorsqu'un composant graphique représente une valeur du modèle à l'écran mais ne permet pas de la modifier. Pour que le composant se mette à jour automatiquement lorsque la valeur du modèle change, il suffit de lier sa propriété à celle du modèle.

4.4.2 Liens unidirectionnels complexes

Les liens unidirectionnels simples décrits ci-dessus permettent de s'assurer qu'une propriété a toujours la même valeur qu'une autre. Ils représentent donc, en quelque sorte, une équation (ou contrainte) d'égalité entre deux valeurs, de la forme :

$$v_1 = v_2$$

Toutefois, cette équation a une direction, dans la mesure où seule la valeur de droite peut changer « de son plein gré », la valeur de gauche se contentant de la suivre.

Il est parfois intéressant d'avoir des équations plus complexes entre deux valeurs. Par exemple, si p représente le périmètre d'un cercle et r son rayon, l'équation suivante exprime la manière dont le périmètre se calcule en fonction du rayon :

$$p = 2\pi\, r$$

JavaFX permet également d'exprimer ce genre de liens unidirectionnels plus complexes entre différentes propriétés. L'exemple ci-dessous l'illustre.

```
DoubleProperty r = new SimpleDoubleProperty();
DoubleBinding p = r.multiply(2d * Math.PI);

p.addListener((o, oV, nV) -> System.out.println(nV));
r.set(0.5);                     // affiche …3.1415
r.set(1);                       // affiche …6.2831
```

4.4.3 Liens bidirectionnels

Avec un lien bidirectionnel, deux propriétés s'observent mutuellement et, dès que l'une change, l'autre est également changée pour qu'elle ait la même valeur.

Un lien bidirectionnel peut être établi entre deux propriétés au moyen de la méthode `bindBidirectionnal`, similaire à la méthode `bind`. Au moment de l'établissement du lien bidirectionnel, la valeur de la *seconde* propriété (passée en argument) est copiée dans la première, mais après cela les deux propriétés sont traitées de manière totalement symétrique : la modification de l'une des deux provoque la modification de l'autre. L'exemple ci-dessous illustre ce comportement.

```
ObjectProperty<String> p1 = new SimpleObjectProperty<>();
ObjectProperty<String> p2 = new SimpleObjectProperty<>();

p1.addListener((o, oV, nV) -> System.out.println(nV));

p2.set(”hello”);                // n'affiche rien
System.out.println(”---␣binding␣p1␣to␣p2”);
p1.bindBidirectional(p2);       // affiche ”hello”
p2.set(”world”);                // affiche ”world”
p1.set(”bonjour”);              // affiche ”bonjour”
```

Un lien bidirectionnel est utile p.ex. lorsqu'un composant graphique représente une valeur du modèle à l'écran et permet également sa modification. Pour que le composant se mette à jour si la valeur du modèle change, et que le modèle soit mis à jour dès que l'utilisateur interagit avec le composant, un lien bidirectionnel peut être établi entre la propriété du modèle et celle du composant.

5 Gestion des événements

Un programme doté d'une interface graphique ne fait généralement qu'attendre que l'utilisateur interagisse avec celle-ci, réagit en conséquence, puis se remet à attendre. Chaque fois que le programme est forcé de réagir, on dit qu'un **événement** (*event*) s'est produit.

Cette caractéristique des programmes graphiques induit un style de programmation particulier nommé **programmation événementielle** (*event-driven programming*). Ce style se retrouve également dans toutes les applications dont le but principal est de réagir à des événements externes, p.ex. les serveurs (Web et autres).

Au cœur de tout programme événementiel se trouve une boucle traitant successivement les événements dans leur ordre d'arrivée. Cette boucle, nommée **boucle événementielle** (*event loop*), pourrait s'exprimer ainsi en pseudo-code :

```
tant que le programme n'est pas terminé
  attendre le prochain événement
```

```
traiter cet événement
```

Cette boucle est souvent fournie par une bibliothèque et ne doit dans ce cas pas être écrite explicitement. Il en va ainsi de la plupart des bibliothèques de gestion d'interfaces graphiques, dont JavaFX.

Si la boucle événementielle peut être identique pour tous les programmes, la manière dont les différents événements sont gérés est bien entendu spécifique à chaque programme. Dès lors, il doit être possible de définir la manière de traiter chaque événement.

Cela se fait généralement en écrivant, pour chaque événement devant être traité par l'application, un morceau de code le traitant, appelé le **gestionnaire d'événement** (*event handler*). Ce gestionnaire est associé, d'une manière ou d'une autre, à l'événement.

La boucle événementielle est séquentielle, dans le sens où un événement ne peut être traité que lorsque le traitement de l'événement précédent est terminé. Dans le cas d'une interface graphique, cela signifie que celle-ci est totalement bloquée tant et aussi longtemps qu'un événement est en cours de traitement. Pour cette raison, les gestionnaires d'événements doivent autant que possible s'exécuter rapidement. Si cela n'est pas possible, il faut utiliser la concurrence pour effectuer les longs traitements en tâche de fond — ce qui sort du cadre de ce cours.

5.1 Gestion des événements dans JavaFX

Dans une application graphique JavaFX, la boucle événementielle s'exécute dans un fil d'exécution (*thread*) séparé, nommé le **fil d'application JavaFX** (*JavaFX Application Thread*). Ce fil d'exécution est démarré automatiquement au lancement d'une application JavaFX, rendant la boucle événementielle invisible au programmeur. Elle n'en existe pas moins !

L'ajout par JavaFX d'un fil d'exécution séparé introduit malheureusement tous les problèmes typiques liés à la programmation concurrente, raison pour laquelle il est important d'obéir à quelques règles lors de l'écriture d'une application JavaFX. En particulier, toute manipulation de nœuds JavaFX faisant partie d'un graphe déjà attaché à un objet de type `Scene` doit se faire depuis le fil d'application JavaFX, de même que toute création d'objet de type `Scene` ou `Stage`.

Dans JavaFX, les gestionnaires d'événements sont généralement des objets implémentant l'interface fonctionnelle et générique `EventHandler`. Le paramètre de type de cette interface spécifie le type de l'événement géré par le gestionnaire. Les gestionnaires d'événements sont attachés à une source d'événements — souvent un nœud — et leurs méthodes sont appelées chaque fois qu'un événement se produit. Les gestionnaires sont donc assez similaires aux observateurs du patron *Observer*, la méthode `update` de ces derniers gérant l'événement « l'état du sujet a changé ».

La classe ci-dessous illustre la structure typique d'un programme JavaFX. Comme toute classe principale d'une application JavaFX, elle hérite de la classe `Application` et créée l'interface graphique dans la méthode `start`.

```
public final class JavaFXApp extends Application {
  public static void main(String[] args) {
    launch(args);
  }

  @Override
  public void start(Stage primaryStage) throws Exception {
    Button button = new Button("click␣me!");
    button.setOnAction(e -> System.out.println("click"));

    Scene scene = new Scene(button);
    primaryStage.setScene(scene);
    primaryStage.show();
  }
}
```

La méthode `main` est appelée sur le fil d'exécution principal du programme, comme toujours en Java. Dans la terminologie JavaFX, ce fil est appelé le fil de lancement (*launcher thread*).

La méthode `main` se contente d'appeler la méthode `launch`, qui démarre (entre autres) le fil d'application JavaFX, puis appelle la méthode `start` sur ce fil. C'est la raison pour laquelle la création de l'interface doit se faire dans la méthode `start` et pas dans la méthode `main` ou dans le constructeur de la classe.

Lorsque l'utilisateur clique sur le bouton, la méthode du gestionnaire attaché au bouton via `setOnAction` est également exécutée sur le fil de l'application JavaFX.

La figure 4 illustre l'exécution des différentes parties du programme sur les deux fils existants.

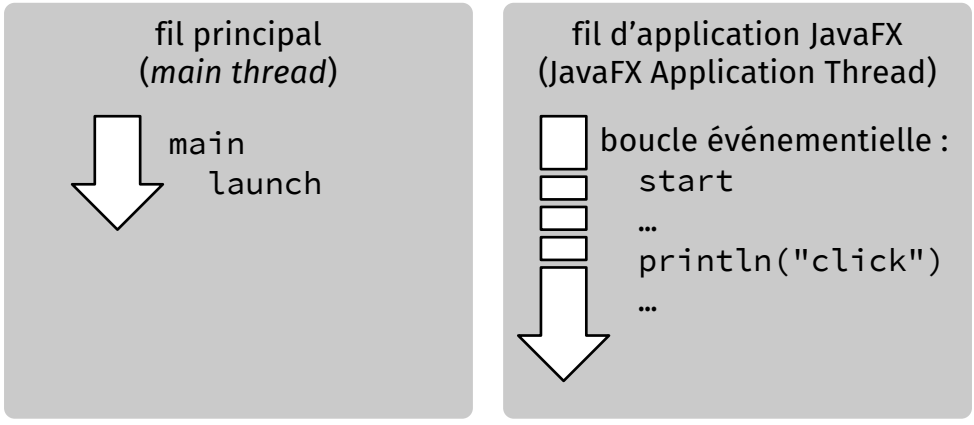

Fig. 4: Fils d'exécution d'un programme JavaFX simple

Lorsqu'un événement se produit, JavaFX collecte les informations à son sujet dans un

objet passé au gestionnaire. Par un léger abus de langage, cet objet lui-même est nommé événement (*event*).

JavaFX définit un grand nombre de types d'événements, parmi lesquels :

- les événements liés à la souris (*mouse event*) : clic d'un bouton, déplacement du pointeur, survol, etc.
- les événements liés à un écran tactile : appui du doigt, glissement du doigt dans une direction donnée, etc.
- les événements liés au clavier : pression et relâchement d'une touche, etc.

En plus de ces événements généraux, des événements plus spécialisés sont également générés par différents composants. Par exemple, un bouton génère un événement lorsqu'il est pressé, et ainsi de suite.

6 Références

- *Pro JavaFX 8* de Johan Vos, Weiqi Gao, Stephen Chin, Dean Iverson et James Weaver,
- Java Platform, client technologies (la partie consacrée à JavaFX).
- la documentation de l'API JavaFX, en particulier les paquetages, classes et interfaces suivants :
  - la classe `Application`,
  - la classe `Stage`,
  - la classe `Scene`,
  - la classe `Node` et ses différentes sous-classes.