TP4 événements : Event et Handler

Attention lorsque vous programmez, apportez un soin tout particulier à la «propreté» de votre code : déclarez ce qu'il faut à l'endroit qu'il faut, utilisez à bon escient l'héritage et les variables statiques, créez les classes et les méthodes qu'il faut de manière à éviter la duplication de code, etc...

Gardez toujours une fenêtre ouverte sur la documentation JavaFX de la classe que je vous demande d'utiliser, ou que vous manipulez. Il faut absolument balayer cette documentation, vous aurez un aperçu des possibilités de la classe, et ensuite vous trouverez plus facilement les propriétés et méthodes dont vous aurez besoin. Vous vous repèrerez également plus facilement dans la hierarchie des classes javaFX.

C'est la dernière fois que je mets ce message. Ne l'oubliez pas pour les prochains TP. `:)`

Manipulations de base, filters, handler.

Pour les deux premiers exercices, je vous demande de procéder comme suit pour gérer vos événements :

1. d'abord écrire une classe *Ecouteur* implémentant l'interface EventHandler,
2. ensuite instancier les écouteurs, des objets de cette classe,
3. et enfin faire les abonnements de ces écouteurs aux événements des composants.

Pour les exercices suivants, procédez comme vous le souhaitez : vous pourrez par exemple les déclarer à la volée en classe locale anonyme lors de l'appel de `addEventHandler` (ou `addEventFilter`) ou utiliser une λ-expression dans une méthode `setOn`*xxx*.

Exercice 1 parcours d'un évènement dans le graphe de scene : `addEventHandler`, `addEventFilter`, `consume`.

1. Écrire une application `TestActionEvent`. Cette application doit contenir : un `Pane` comme panneau racine (on l'appelera `root` "as usual") ; un cercle de diamètre 50 que vous placerez où vous le souhaitez ; deux boutons `b1` et `b2` que vous placerez où vous le souhaitez ; un label `labMessage` assez grand.

2. Nous allons maintenant créer une classe `Ecouteur` gestionnaire d'évènements, cet écouteur se contentera pour le moment d'afficher un message en sortie.

   • Rajoutez donc (soit en "innerclass" de `TestActionEvent`, soit dans un fichier séparé) une classe `Ecouteur` implémentant l'interface `EventHandler`. Le constructeur de cette classe sera `Ecouteur(String message)`. Faire en sorte que la méthode `public void handle(Event e)` affiche `message` ainsi que le type d'événement, le noeud cible, et le noeud source. Quelles sont les méthodes de la classe Event permettant de faire cela ?

   • Instanciez un écouteur avec le message `"Bouton Action 4 "`. Abonnez cet écouteur au bouton avec la méthode `addEventHander`, il faudra dire que cet écouteur est appelé sur un événement `ActionEvent.ACTION` (Quelle est la classe mère de ActionEvent ?) Tester.

3. Laisser l'écouteur affichant `"Bouton Action 4"` en place pour cette question. Deux méthodes permettent d'abonner un écouteur aux événements sur un composant : `addEventHandler` et `addEventFilter`. Nous allons placer sur différents composants deux écouteurs avec ces deux méthodes différentes. Sur chacun des composants (le bouton `b1`, le panneau `root` et la scène), posons un écouteur comme "Handler" et un écouteur comme "Filter".

   • Il faut donc instancier 6 autres écouteurs, avec des messages différents (du genre `"Je suis le filter du bouton b1"`) qui vous permettront de savoir qui est qui), et abonner le bouton `b1`, le panneau `root` et la scène à deux écouteurs chacun (un comme Filter, un comme Handler).

   • Tester lors d'un clic sur la fenêtre, le panneau, et le bouton b1. Quel est le parcours de l'événement dans le graphe de scène ?

   • Essayez d'abonner plusieurs écouteurs comme "handler" sur un même composant, est-ce possible ?

4. Dans la classe `Event`, la méthode `consume()`, permet d'arrêter la propagation de l'évènement.

   • Modifier la classe `Ecouteur` de manière à ce que le constructeur soit `Ecouteur (String message, boolean stop)`. Si `stop` est vrai, la méthode `handle` appelle la méthode `consume` sur l'événement.

   • On garde les 3 objets (scene, root, et bouton) abonnés chacun à 2 écouteurs, l'un comme "handler", et l'autre comme "filter". Dans chacun des cas suivants, comment choisir les différents paramètres `stop` des 6 écouteurs utilisés pour que :
   - le filtre de *root* soit exécuté, mais pas celui du bouton ;
   - le "handler de de *root* soit exécuté, mais pas celui de la scène ;
   - le "handler de de *root* soit exécuté, mais pas celui du bouton `:D`.

Exercice 2 `setOn`*xxx* et écouteurs partagés.

Faire une copie de l'application précédente en supprimant tous les écouteurs sauf les deux (normalement, il doit y en avoir 2) handlers du bouton `b1` (vous vérifiez bien que le paramètre stop de ces écouteurs est à false).

1. Utiliser la méthode `setOnAction` pour positionner un $3^{ème}$ écouteur sur `b1`.
   - Dans quel ordre les écouteurs sont-ils exécutés ?
   - Est-il possible de positionner deux écouteurs différents sur le même bouton en utilisant `setOnAction` ?
2. Modifier votre classe `Ecouteur` de manière à ce que la méthode `handle` affiche dans `labMessage` le texte : `"je suis le bouton b1"`, `"je suis le bouton b2"`, `"je suis le cercle"`, `"Je suis le panneau` ou `"je suis le label"` suivant la source de l'événement.

   Si votre `Ecouteur` n'est pas une "innerClass", il faudra faire en sorte que votre écouteur "connaisse" ces différents composants (en les passant en paramètre lors du constructeur par exemple).
3. Créer un seul écouteur, et abonner cet écouteur aux `ActionEvent.ACTION` sur chacun des boutons, le cercle, le panneau, et le label. Est-il possible de faire ceci avec la méthode `setOnAction` quel que soit le type du composant ? Tester votre application en cliquant sur chacun des composants. Tous les composants peuvent-il déclencher un ActionEvent ?
4. Nous allons faire réagir l'application aux clics de souris, cela peut se faire en utilisant la méthode `setOnMouseClicked` ou l'appel de `addEventHandler` avec en 1er paramètre un `MouseEvent.MOUSE_CLICKED`.
   - Quelles sont les classes parentes d'un `MouseEvent` ?
   - Écrire une classe `EcouteurDeClic` (qui comme tout écouteur implémentera l'interface `EventHandler`) ; la méthode `handle` affichera dans `labMessage` la position en X et Y ou a eu lieu le clic de la souris. Attention : rappelons que la méthode `handle` prend un paramètre `Event` en entrée, mais que les méthodes `getX`, et `getY` qui permettent de récupérer la position d'un click sont des méthodes de `MouseEvent`.
   - Instancier un écouteur de clic, et l'abonner aux clics de souris du label, du cercle et du `root`. La position retournée par ces méthodes `getX` et `getY` est elle dans le repère du composant, ou de la fenêtre ?

Exercice 3 différentes façons d'écrire/abonner un écouteur.

Dans les exercices suivants, on aura souvent besoin d'avoir une fenêtre avec un bouton quit placé en bas à droite. Cet exercice consiste à écrire plusieurs versions d'une classe `BorderWithQuit`, cette classe héritera de `BorderPane`, elle aura un bouton *quit* placé en bas à droite. Un clic sur ce bouton fermera l'application. Je rappelle qu'il faut utiliser la méthode `Platform.exit()` lorsqu'on quitte une application javaFX. Cela permet en particulier de sortir proprement de votre application javaFX, en appelant la méthode `stop()` que vous avez peut-être redéfinie pour fermer une base de donnée, ou autre connexion.

Vous testerez ces différentes versions dans une application ayant un BorderWithQuit comme racine.

1. Ecrire 3 versions de cette classe différent seulement par la manière dont est créé l'écouteur du bouton quit :
   (a) `BorderWithQuitCLA` où vous utilisez une *C*lasse *L*ocale *A*nonyme lors de l'appel de `addEventHandler` avec un `ActionEvent`
   (b) `BorderWithQuitLA` où vous utilisez une expression *LA*mbda lors de l'appel de `setOnAction`.
   (c) `BorderWithQuitEH` version où le `BorderWithQuitEH` lui même implémente l'interface *EventHandler*.
2. Que choisiriez-vous, et pourquoi, pour écrire/abonner un écouteur si :
   - l'écriture de `handle(..)` est complexe, demande par exemple l'écriture de sous routines, accède à des objets externes à la classe où il est utilisé ; et le programme doit utiliser plusieurs instance et l'on utilise plusieurs instances différentes de de cette classe Écouteur ?
   - chaque composant a sa propre instance d'écouteur ; la méthode `handler` ne contient que quelques instructions, et n'accéde qu'à des membres locaux à la classe ?
   - chaque composant doit être abonné (pour le même évènement) à plusieurs écouteurs différents ?

Exercice 4 première application avec `ActionEvent` et `MouseEvent`.

1. Écrire une application dont la racine soit un des `BorderWithQuit` de l'exercice précédent. Cette application contiendra dans une fenêtre 400×400 : en haut, un bouton $+5$, un bouton -5 et un label qui affichera 10 au départ ; en bas le bouton "quit" ; au centre un `Pane` *pc* contenant un cercle de 10 pixels de rayon, centré (à peu près).

2. L'application doit assurer en permanence que le rayon du cercle soit égal à l'entier affiché dans le label, le rayon ne pouvant évidemment devenir négatif. Une action sur :
   - le bouton $+5$ incrémente de 5 l'entier affiché dans le label ainsi que le rayon du cercle,
   - le bouton -5 décrémente d'autant l'entier affiché dans le label ainsi que le rayon du cercle.

3. Faire en sorte maintenant qu'un clic sur le pannel *pc* déplace le centre du cercle à l'endroit du clic.

4. Quels événements et méthodes utiliser pour déplacer le disque avec un drag and drop ?

Exercice 5 utilisation de grille et de boutons.

1. Définir une classe publique, nommée `MyButton` , dérivant de `Button` , encapsulant deux attributs entiers, `line` et `column` , et disposant des construtteurs et méthodes suivantes :

   - `MyButton(int line, int column)` crée une instance de MyButton et définit les valeurs des deux attributs `line` et `column` comme on le pense, et l'étiquette du bouton est le texte `"(line,column)"` ;
   - `void setLine(int line)` et `void setColumn(int column)` qui affectent les valeurs respectives des attributs `line` et `column` ;
   - `int getLine()` et `int getColumn()` fournissent respectivement la valeur de l'attribut `line` et celle de l'attribut `column`.

   Pas de méthode main pour cette classe `MyButton`.

2. Écrire une classe, `MyButtonTest`, qui dérive de `Scene` et dont une instance contient un bouton de type `MyButton` dont les attributs `line` et `column` ont pour valeur deux entiers récupérés sur la ligne de commande.

   Cette scène contient de plus un `Label` et un bouton `Quit` (cela veut dire qu'il faudra utiliser la classe `BorderWithQuit` que vous avez écrite dans l'exercice 3). Lorsque l'on clique sur le bouton `MyButton`, le nombre `line + column` est affiché comme texte du Label. Vous testerez cette classe `MyButtonTest` pour vérifier la conformité de la lasse `MyButton` à ses spécfications.

3. Définir une classe, nommé `MyPanel` qui dérive de... , dont le constructeur est :
   • `MyPanel(int nlines, int ncolumns)` qui crée une instance de la classe ; ce panneau est rempli d'une grille ayant `nlines` lignes et `ncolumns` colonnes, et dans chaque case se trouve une instance de `MyButton` dont l'attribut `line` représente le numéro de la ligne de la grille où se trouve le bouton, idem pour l'attribut `column` et le numéro de la colonne, les lignes étant numérotées du haut vers le bas et les colonnes de gauche à droite.

   L'unique méthode de la classe MyPanel est :
   `MyButton getButton (int line, int column)` qui donne un accès sur le bouton qui se trouve à la ligne `line` et à la colonne `column`.

4. Tester en écrivant une application `MyPanelTest` qui contienne un `MyPanel` de 3 lignes et 5 colonnes, et toujours le bouton Quit.

5. Écrire une classe `MyButtonPanel` qui dérive de `MyPanel` et qui ajoute à chaque bouton un gestionnaire d'événements de sorte que lorsque l'on clique sur l'un des boutons de la grille, une fenêtre popup apparaît et affiche les valeurs de `line` et `column` du bouton cliqué.

   Pour ouvrir une nouvelle fenêtre, il est possible de déclarer une nouvelle `Stage` *dans la méthode* `start`*, la construire comme vous avez fait pour la fenêtre principale et l'ouvrir avec la méthode show(). Mais généralement, pour utiliser des fenêtres popup, on utilise la classe* `Dialog`*. La sous classe* `Alert` *est une sous-classe de* `Dialog` *prévue pour les messages d'informations ou de confirmation simples. Il suffit dans notre cas de :*

   – *créer votre alerte en choisissant son type*
   ```
   Alert myPopUp = new Alert(AlertType.INFORMATION)
   ```

– *choisir le titre et le message, avec*
```
myPopUp.setTitle("MonTitre");
myPopUp.setContentText("J'ai mal aux dents");
```
– *montrer la fenêtre avec* `myPopup.show()`, *ou* `myPopUp.showAndWait();`

Cela ouvrira une fenêtre avec un bouton OK pour la fermer.

La classe `Dialog` *permet de faire des manipulations plus compliquées, par exemple demander à l'utilisateur de faire des choix, je vous laisse regarder la documentation pour quand vous en aurez besoin.*

• Faire une deuxième version où l'écouteur affiche la position en X et la position en Y dans deux `Alert différents`. Tester cette version en appelant pour montrer ces popup la méthode `show()`, puis recommencer avec la méthode `showAndWait()`.

Quelle est la différence entre `show()`, and `showAndWait()` ?

6. Reprendre la question 4 pour tester maintenant la classe `MyButtonPanel`

7. Reprendre les questions 1 et 2 pour arriver à construire une grille de boutons (qui sont des instances d'une sous-classe de `MyButton` , cette sous-classe ne devant pas être interne à une autre classe) qui, lorsqu'on les actionne, affichent leurs coordonnées dans un Label situé dans la zone nord de la fenêtre principale.

Exercice 6 manipulations élémentaires de boutons.

Nous fournissons 2 images `TP_04_play.png` et `TP_04_pause.png` sur UPdago. Ces images vous permettront de construire deux icônes à placer sur les boutons.

Se reporter à la question 2 de l'exercice 8 du TP2 pour la manipulation. Même si cela n'est pas forcément le plus propre, mettre le répertoire Icons *à l'endroit même où se trouve le fichier java qui l'utilise, cela simplifie l'écriture du chemin à mettre dans l'appel de passer en paramètre à* `getResourcesAsStream(`*chemin_acces_image*`)`.

Si ces icônes étaient trop grandes à votre goût, choisir le constructeur d'`Image` avec les paramètres permettant de redimensionner l'image.

1. Placez dans une fenêtre deux instances de `Button`, décorées omme on le voit sur la partie gauche de la figure 1 ; les deux instances sont dans un FLowPane.

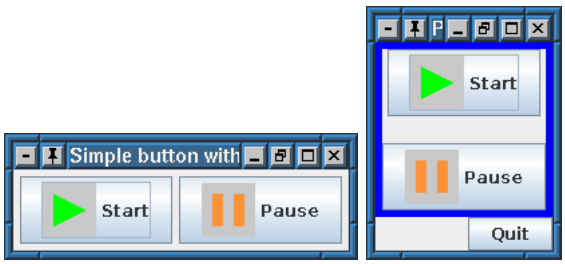

Figure 1: Les différents aspects des fenêtres demandées

2. On place maintenant les deux instances de boutons, comme on le voit sur la partie droite de la figure 1. Un bouton étiqueté "Quit" est dans la fenêtre principale et a pour effet de terminer l'application. On soignera tout particulièrement le placement des deux boutons.

3. On définit maintenant une nouvelle classe, que l'on nomme `ButtonStartPause` ; cette classe dérive de la classe `Button`, et enrichit cette classe en définissant un nouveau type de bouton dans lequel on ne peut pas mettre de texte, mais où figure une icône, qui est initialement la petite flèche verte. Lorsque l'on clique sur le bouton, il pourrait se produire beaucoup de choses, déterminées par un traitement de l'événement `ActionEvent` engendré par le clic sur le bouton mais, de surcroît, l'icône sur le bouton change, et commute du symbole de démarrage au symbole de pause. La classe n'exporte qu'un seul constructeur, `ButtonStartPause (int size)` , qui crée un bouton portant un icone dont la taille a pour coté `size`. Pour tester le comportement de ce nouveau composant, on placera une instance de taille 40 pixels de la classe `ButtonStartPause` dans la zone sud d'un `BorderPane` , et une autre instan e, de taille 100 pixels, dans la zone entrale de ce même `BorderPane`. Lorsque l'on actionnera la première instance (celle qui est dans la zone sud), le programme se terminera ; et lorsque l'on actionnera la seconde instance (celle qui est dans la zone entrale), un petit message sera affiché dans la console.

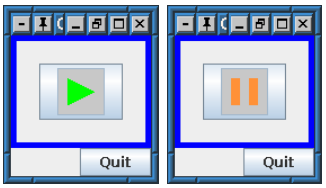

Figure 2: Les deux aspects du `ButtonStartPause`

Exercice 7 changements de couleur et clics de souris.

Dans cet exercice, on étudie quelques manières de changer la couleur en fonction d'événements intervenant sur les composants.

1. Écrivez une petite application qui place dans une fenêtre un Panel dont le fond est colorié d'une couleur définie en bleu par l'application.

   Rappel : on peut changer la couleur du fond d'un label ou panneau `c` :
   – soit en utilisant la méthode `setStyle` (avec un paramètre String "à la CSS") avec `c.setStyle("-fx-background-color:blue;")`
   – soit en utilisant la méthode `setBackground`, à la quelle on passe en paramètre un `Background` créé lui à partir d'un `BackgroundFill` comme on l'a déja vu avec
   `c.setBackground(new Background(new BackgroundFill(Color.BLUE,CornerRadii.EMPTY,Insets.EMPTY)));`

   Sur `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html`, vous trouverez une référence pour tous les attributs CSS que l'on peut utiliser dans `setStyle` (en particulier `"-fx-text-fill"` pour choisir la couleur d'un texte).

2. Écrire une application qui place dans une fenêtre, un label colorié par deux couleurs : l'une pour le fond du Label, l'autre pour le texte qui est placé dans le Label. La couleur du texte peut être positionnée avec `setStyle` ou bien avec la méthode `setTextFill`.

3. Reprendre l'application de la première question 1, mais définir deux couleurs pour le fond du Panel ; l'application créée doit changer la couleur de fond lorsque l'on clique sur la souris (quel que soit le bouton) et passer, au fur et à mesure des clics, successivement d'une couleur à l'autre. Plusieurs problèmes, que l'on résout en ajoutant des attributs liés au JPanel : la définition des deux couleurs, la commutation d'une couleur à une autre (il est possible de faire cela sans test), et la gestion des clics de souris.

4. Reprendre le même principe pour les changements de couleur d'un JLabel : quand on clique sur la souris dans le JLabel, ce dernier échange sa couleur de fond avec sa couleur de texte.

Exercice 8 utilisation de la molette de la souris.

Les événements générés par la molette de la souris sont des `ScrollEvent`[1] Plusieurs méthodes permettent d'avoir des informations sur cet événements. En particulier `getDeltaY()` permet d'avoir le déplacement de la molette ; comme d'habitude, il y a moyen de savoir si une touche *ALT*, *CONTROL* ou autre modifieur est appuyée pendant le mouvement.

1. Écrire une application qui contienne un `Label compteur`, ce label étant programmé pour réagir aux actions de la molette de la souris : `compteur` affiche un entier, initialement 0 ; cet entier augmente ou diminue suivant le sens dans lequel on tourne la molette.

2. Modifier l'application précédente de manière à ce que l'appui sur la touche *SHIFT* pendant l'action sur la molette multiplie par 10 la vitesse à la quelle change le compteur.

[1] Attention. Si en Swing ou en AWT, bouger la molette de la souris est un `MouseEvent`, ce n'est pas le cas en javaFX. En javaFX, un `ScrollEvent` n'est pas un `MouseEvent`, un `ScrollEvent` est plus général et peut provenir également d'un touchpad, ou d'un écran tactile, etc...

Exercice 9 communication entre fenêtres.

On vous demande d'écrire une application qui ouvre deux fenêtres. Cela peut se faire en déclarant une `Stage autreStage` dans la méthode start de l'application principale. Il suffit alors de construire `autreStage` et de terminer par `autreStage.show()`.

1. Écrire une classe `SecondeFenetre` héritant de `Stage`, cette classe contiendra un TextField `tfSecondaire`. Elle fournira les méthodes :
   – `void afficheMessage(String m)` qui affichera la chaine `m` dans `tfSecondaire`
   – `void setOnTextChanged(EventHandler eh)` qui abonnera l'EventHandler `eh` aux changements de texte sur `tfSecondaire`

2. Faire en sorte que l'application ouvre une fenêtre principale comme d'habitude, et une `SecondeFenetre`, la fenêtre principale contiendra un `Label lSecondOuvert`, un `TextField tfPrincipal`, et un bouton `Envoi`.

3. Maintenant, faire en sorte que :
   - lorsque l'utilisateur actionne le bouton `Envoi`, le texte de `tfPrincipal` est recopié dans `tfSecondaire`.
   - lorsque l'utilisateur modifie le texte du `tfSecondaire`, celui ci est recopié dans le `tfPrincipal`.
   - le label `lSecondOuvert` indique si la seconde fenêtre est ouverte ou fermée.

Exercice 10 déplacement d'un joueur.

Reprendre l'exercice 8 du TP2. Ecrire une application qui contienne : un `Board` et un panneau `Direction`. Initialement, un joueur que vous représenterez par le charactère `'O'` sur fond bleu est à la position (5,5). Le panneau de direction permettra de déplacer ce joueur sur le damier.