Développement de programme dans un environnement graphique Automne 2022 Cégep Limoilou Département d'informatique Professeur : Martin Simoneau	**Formatif 3** ***Gestion d'événements*** ***Menu***

Objectifs

- Comprendre et pratiquer la gestion des événements *javaFX*
- Programmez des menus.

À remettre :

- À faire seul.

Contexte :

- Remettre votre projet sur Léa
- Cet exercice sert à assimiler les concepts qui seront nécessaires pour le TP2.

Événements

1. Ordre d'appel des Filter et handler

- On veut identifier tous les noeuds qui vont traiter un même événement. La source est le *noeud* sur lequel est présentement appelé l'événement et le target est le nœud le plus haut qui reçoit l'événement. La source change, mais pas le target. Assurez de bien voir ce comportement dans vos résultats. C'est important de bien le comprendre !
- Programmez le gestionnaires ***showEventInfo*** dans la méthode *start* de la classe *App1* à l'aide d'un callback. Le code est préparé pour faire une closure, vous pouvez le changer en classe interne ou anonyme si ça vous aide.
    - le gestionnaire doit afficher
        - le bouton primaire ou secondaire -> avec *event.getButton()*
        - La cible de l'événement () -> avec *event.getTarget()*
        - La source de l'événement () -> avec *event.getSource()*

Exemple :

```
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.VBox et sur la cible class javafx.scene.shape.Polygon
```

    - Pour obtenir la classe, il suffit d'appeler la méthode *getClass*() sur un objet.
- En utilisant ce gestionnaire, on va pouvoir observer la chaîne d'appels d'un événement. Pour cela il faut brancher le gestionnaire sur chaque nœud. Programmez la méthode ***ajouteFiltreEvenement*** ().
    - Cette méthode reçoit :
        - Le handler (ici *showEventInfo*)
        - Le type d'événement à filtrer (*MouseEvent.Mouse_Clicked*)
        - Les nœuds sur lesquels appliquer le gestionnaire d'événements.

- La méthode doit balayer chaque nœud et lui associer le *handler* reçu en paramètre avec la méthode *node.addEventFilter()*.

- Dans la méthode *start*(), appelez la méthode ***ajouteFiltreEvenement*** que vous venez de programmer en lui passant tous les nœuds : *root, topHBox, bottomHBox, circle, rect, img et poly.*

  - Lancez l'application et cliquez sur les différentes parties de la fenêtre. À chaque clic vous devriez apercevoir la chaîne de filtrage de l'événement de la racine jusqu'à la cible. Par exemple, si l'on clique sur le polygone on obtiendra :

```
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.VBox et sur la cible class javafx.scene.shape.Polygon
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.HBox et sur la cible class javafx.scene.shape.Polygon
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.shape.Polygon et sur la cible class javafx.scene.shape.Polygon
```

  - Programmez maintenant la méthode *ajouteGestionnaireEvenement* qui fait la même chose que *ajouteFiltreEvenement* (avec les mêmes arguments), mais en utilisant *node.addEventHandler*() au lieu de *node.addEventFilter*().

  - Associez maintenant *showEventInfo à chaque nœud avec la méthode ajouteGestionnaireEvenement().*

Lancez l'application. Cette fois-ci, si vous cliquez sur le cercle, vous verrez apparaître le texte suivant :

```
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.VBox et sur la cible class javafx.scene.shape.Circle
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.HBox et sur la cible class javafx.scene.shape.Circle
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.shape.Circle et sur la cible class javafx.scene.shape.Circle
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.shape.Circle et sur la cible class javafx.scene.shape.Circle
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.HBox et sur la cible class javafx.scene.shape.Circle
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.VBox et sur la cible class javafx.scene.shape.Circle
```

  - Observez donc que le filtrage se fait toujours de la racine vers la cible. Après le filtrage on voit la transmission de l'événement de la cible vers la racine (*bubbling*). Observez également que la cible (*target*) est toujours la même alors que la source qui représente l'objet sur lequel le gestionnaire est déclenché change.

  - Notez également que si vous appuyez avec le bouton droit vous obtiendrez *le bouton SECONDARY* au lieu du *bouton PRIMARY* en début de message.

2. Arrêter la propagation

- Est-ce que la propagation de l'événement passe toujours toute la chaîne ? Non, il est possible d'interrompre la transmission de l'événement en appelant la méthode ***consume*** sur l'événement.
- Ajoutez le ***e.consume*** à la fin du gestionnaire d'événement *showEventInfo.* Vous obtiendrez alors une seule ligne par clic puisque l'événement cesse de se propager après le premier filtre (sur root).

```
le bouton PRIMARY a été enfoncé sur la source class javafx.scene.layout.VBox et sur la cible class javafx.scene.shape.Circle
```

3. UserData très utile

- *JavaFX* définit un champ nommé *userData* sur chaque *nœud*. Vous êtes libre d'y placer l'information que vous voulez. Ce champ est très pratique.
- Commencez par retirer les commentaires incluant *setUserData* dans la méthode *createContent*(). Vous allez ainsi ajouter une chaîne de caractères sur chaque nœud indiquant sa(ses) couleur(s).
- On va l'utiliser pour afficher la couleur de chaque nœud. Programmez maintenant le gestionnaire ***showTargetColor*** qui affiche dans la console.

```
la couleur de la cible est light green et la couleur de la source est light green
```

    - Utilisez *event.getSource().getUserData()* pour retrouver la couleur qu'on a placée dans la méthode *createContent*().
- Commentez l'appel de *ajouteFiltreEvenement* et dans l'appel de *ajouteGestionnaireEvenement* , remplacez *showEventInfo* pour *showTargetColor*.
- Lancez l'application et vous devriez alors voir la console afficher la couleur qu'on a placée avec *setUserData*.
- Notez qu'il existe une façon pour ajouter d'autres propriétés à volonté directement à partir de la *map* de propriétés :

```
Node node = …
node.getProperties().put("foo", "bar");
…
Object foo = node.getProperties().get("foo");
```

4. Où a lieu l'événement

- On va maintenant programmer le gestionnaire d'événements ***whereHandler*** qui indique l'emplacement où a eu lieu le *clic*, d'abord par rapport à la scène puis par rapport au nœud lui-même. Programmez le code de ce gestionnaire pour qu'il affiche :

```
l'événement a eu lieu dans la scène a (scene x, scene y)
l'événement a eu lieu sur la cible a (node x, node y)
```

- Pour y arriver utilisez
    - *event.getSceneX()* et *event.getSceneY()*
    - *event.getX()* et *event.getY()*
- Commentez l'appelle de *addEventFilters*
- Dans l'appel de *ajouteGestionnaireEvenement* changez *showEventInfo* pour *whereHandler*.
- Lancez l'application et validez que les positions x et y données correspondent bien à l'endroit où vous avez cliqué. Rappelez-vous que le point 0,0 est dans le coin supérieur gauche et que l'axe des y est positif vers le bas.

5. Pourquoi placer le gestionnaire plus haut dans la chaîne d'appel

- On peut évidemment placer des gestionnaires d'événements directement sur chaque noeud, mais c'est souvent une mauvaise stratégie. On préférera souvent utiliser un gestionnaire plus en amont parce que ce dernier est en mesure de voir le portrait d'ensemble. Pensez à un jeu d'échecs. Est-ce le pion qui a la meilleure vision ou l'échiquier ?
- Pour vérifier cela, on va programmer le gestionnaire d'événements ***identificationHandler***. Ce dernier affiche simplement la classe de la *target* et le type d'événement. Le *MouseEvent* reçu contient les méthodes .*getTarget*() et *getEventType*()

```
Il y a eu un MOUSE_ENTERED_TARGET sur la forme javafx.scene.layout.VBox
```

- Commentez les appels précédents sur *ajouteGestionnaireEvenement* et *ajouteFiltreEvenement.*
- Avec la méthode *addEventHandler, uniquement* sur le nœud racine *root* , ajoutez *l'identificationHandler* pour les événements MouseEv*ent.MOUSE_ENTERED_TARGET* et *MouseEvent.MOUSE_EXITED_TARGET*. (donc 2 appels de *addEventHandler*). Remarquez qu'on peut placer plusieurs gestionnaires sur un même nœud et pour différents événements !

- Lancez l'application et vous verrez afficher tous les endroits où la souris entre ou sort. Pourtant il n'y des gestionnaires que sur le nœud racine. C'est l'endroit où il est plus facile d'observer l'ensemble. En plaçant le gestionnaire au bon endroit, on peut souvent simplifier le code.

6. Ordre des événements hiérarchiques.

- Commentez la programmation des événements faits au numéro 5.
- Un nœud peut avoir simultanément plusieurs gestionnaires d'événements simultanément, mais dans quel ordre sont-ils appelés ? L'ordre dépend d'abord du type d'événement. Les événements forment une hiérarchie. L'élément le plus spécifique (feuille) est d'abord appelé ensuite les éléments plus abstraits (racine). Pour le réaliser, on va enregistrer des *EventHandler* pour différents événements sur le *bouton* qui est à la fin dans l'interface. Dans la méthode ***createGestionairesBouton***, directement avec la méthode addEventHandler, ajoutez les éléments suivants sur le bouton:

Événements	Messages affichés
Event.Any	"any event" + .getEventType()
MouseEvent.ANY	"any mouse " + a.getEventType()
MouseEvent.MOUSE_PRESSED	"mouse pressed"
MouseEvent.MOUSE_RELEASED	"mouse released"

- Lancez l'application.
- Vous allez remarquer qu'aussitôt que votre souris bouge au-dessus du bouton, des affichages ont lieu dans la console… C'est parce que *Event.Any* attrape tous ses événements enfants dont *MOUSE_MOVED*, *MOUSE_ENTERED* et *MOUSE_EXITED*…
- Lorsque vous appuyez sur le bouton, vous allez voir apparaître une série d'affichage en console. (On l'a coloré pour simplifier les explications)

```
1 mouse pressed
2 any mouse MOUSE_PRESSED
3 any event MOUSE_PRESSED
1 mouse released
    1 on action
    2 any event ACTION
2 any mouse MOUSE_RELEASED
3 any event MOUSE_RELEASED
1 any mouse MOUSE_CLICKED
2 any event MOUSE_CLICKED
```

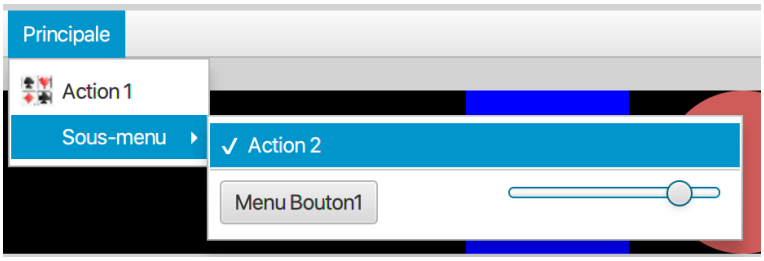
Explications :

1. En jaune on voit que les 3 gestionnaire placé sur le bouton ont répondu au bouton enfoncé.
    - En priorité le gestionnaire associé directement en MOUSE_PRESSED parce que c'est le plus spécifique
    - Ensuite celui associé au MouseEvent.ANY
    - Finalement celui associé au Event.ANY (qui est le plus abstrait des événements)
2. En bleu on a la même chose qu'en jaune mais pour le MouseEvent.Release. Chacun des trois gestionnaires associés au bouton sont appelé dans le même ordre, du plus particulier au plus général.
3. En vert on voit le fameux action *Event* qui est ajouter dans la séquence par javaFX dans la majorité des contrôles pour déclencher une action. C'est le gestionnaire qu'installe la méthode *setOnAction*.
4. Finalement, en rose, le MOUSE_CLICKED est généré suite à un *release* qui est sur la même *target* que le *pressed*.

- On remarque que *on action* survient aussitôt que le mouse release a été détecté. Il est donc prioritaire au MOUSE_CLICKED.

Menus

- Dans la méthode *createMenus* de la classe App1, vous allez créer le menu suivant :



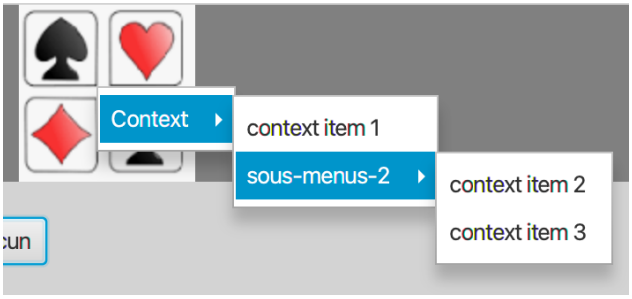
- Créez 4 items de menu
  - *menuItem1* de type *MenuItem* contient le texte ***Action1*** et l'image placée dans *imgView.* L'image peut être installée directement avec le constructeur ou encore avec la méthode *setGraphic*.
  - *menuItem2* est de type *CheckMenuItem* et il contient le texte ***Action 2***. Lorsqu'on le déclenche, il bascule entre l'état sélectionné (Selected) et l'état non sélectionné.
  - *menuItem3* est de type *CustomMenuItem*. Ce type de menu peut contenir n'importe quel nœud. Dans le haut de la méthode, vous avez déjà une variable *menuBox* qui a été programmée. Elle contient un bouton, une étiquette et un slider. Il suffit de passer le nœud menuBox au constructeur pour fabriquer le nœud personnalisé.
  - Instanciez également *separator* de Type *SeparatorMenuItem.*
- Creez 2 menus de type *Menu*
  - *menu* avec le texte *Principal*
  - *sousMenu* avec le texte *Sous-menu*
- Créer *menuBar* de type *MenuBar*
- Assemblez le tout
  - Dans le menubar, mettre :
    - *menuPrincipal*
  - Dans *menuPrincipal*, mettre :
    - *menuItem1*
    - *sousMenu* (remarquez que les menus peuvent être simplement imbriqués comme n'importe quel autre menu item.)
  - Dans *sousMenu*, mettre :
    - *menuItem2*
    - *separator*
    - *menuItem3*
- Finalement, il faut ajouter la *menuBar* comme premier élément de *root* dans la méthode *createContent*.
- Vous pouvez lancer l'application et vérifier que votre menu fonctionne correctement.
- Programmation des comportements des menus dans la méthode *ajouteGestionnaireMenu* :
  - Avec la méthode *menuItem1.setOnAction* faites afficher ***Menu 1 activé*** dans la console.
  - Avec *menuItem2.addEventHandler(…)* faites afficher ***Menu 2 activé*** dans la console. Utilisez l'événement *ActionEvent.ACTION*. Cette syntaxe est équivalente au *setOnAction utilisé avec menuItem1*.
  - On va programmer le bouton du *menuItem3*, *menuBouton* :

- Il doit afficher la valeur du slider : *La valeur est de " + slider.getValue()*
- Il doit également afficher si le menuItem2 est sélectionné. menuItem2.isSelected();
  - "Le menu 2 est sélectionné »
  - "Le menu 2 est non sélectionné »

Exemple :

```
La valeur est de 50.0
Le menu 2 est  sélectionné
```

- Menu contextuel ()
  - Dans la méthode *createContectMenu* de la classe *App1*, on va créer un menu contextuel qui apparaîtra lorsqu'on appuie sur *l'imageView* (avec les symboles de cartes) ou sur le bouton :



    - Vous devez faire le menu et les sous-menus par vous-même.
    - Créez ensuite un objet de type *ContextMenu* et ajoutez à ses items le menu que vous venez de faire.
  - Pour attacher un menu contextuel à un nœud, il y a deux façons de procéder selon le type d'éléments. Pour des nœuds enfants de Control, il y a une méthode très simple que nous verrons plus loin. De façon générale, pour tous les types de nœuds, on peut ajouter un gestionnaire d'événement qui réagit au *MousePressed*.
  - Programmer un gestionnaire d'événement (*setOnMousePressed*) sur l'image :
    - Il vérifie si le bouton droit a été utilisé (l'événement peut vous fournir cette information)
    - Si le bouton de droit a été utilisé, le menu est affiché en utilisant la méthode ***show*** de l'objet *ContextMenu*.La méthode *show* prend en paramètre la fenêtre *ownerWindow* et les coordonnées où doit apparaître le menu contextuel.
      - L'événement de type *ContextMenuEvent* reçu en paramètre peut vous fournir l'endroit du clic avec les méthodes *getScreenX*() et *getScreenY*()
      - Essayez d'utiliser le *root* comme *ownerWindow* du menu contextuel. Ça va créer un problème. Le menu ne se refermera que lorsque vous aurez appuyé sur un item. C'est normal, puisque le menu se ferme lorsqu'on clique à l'extérieur de la classe qui lui sert de propriétaire. Il faut donc choisir comme parent soit :
        - L'image elle-même
        - Le *primaryStage* avec : *root.getScene().getWindow()*
  - *JavaFX* possède également un événement plus direct qui est déclenché uniquement lorsque le bouton de droit est utilisé : *setOnContextMenuRequest*(…). Essayez-la au lieu de *setOnMousePressed.*
  - Finalement, pour les objets enfants de Control, il suffit d'appeler la méthode *setContextMenu* et de lui passer l'objet de type *ContextMenu*. Utilisez cette méthode pour brancher le menu contextuel sur le bouton.
  - Essayez de brancher le menu principal de l'application (celui dans la barre de menu) directement dans un menu contextuel. Ça va créer des problèmes… pourquoi?

FIN