Université de Limoges

R2.02

JavaFX™

Programmation évènementielle

Informations diverses

- M. Synetta VAR
- Adresse de contact : soly.var.aff@unilim.fr
- Support de cours sur Moodle

https://community-iut.unilim.fr/course/view.php?id=1557

Le code d'auto inscription est : **R2.02_2024**

- Ressources sur le réseau :

\\ad.unilim.fr\Pedagogie\pedago-iut\info\Public\ENSEIGNEMENT\BUT1\R2_02

Sommaire

- Rappels…
- La programmation évènementielle
- Le design pattern Observer
- Les évènements dans JavaFX
  - Les types d'évènements
  - La propagation de ces évènements
- Les écouteurs
  - Les Event Filters (les différentes façons de les instancier)
  - Les Event Handlers
  - Les "convenience methods"

Déjà vu

- JavaFX permet la réalisation d'IHM en langage java
- C'est un client « lourd »
- Découpage en 3 couches, modèle de conception MVC
- Les composants sont contenus dans le graphe de scène
- Les composants graphiques se nomment des UI Controls
- L'agencement des nœuds est géré par des containeurs nommés Layout Panes

- On commence la réalisation d'une IHM :
  - En dessinant
  - En cherchant les Layout Panes et les UI Controls les plus adaptés

Programmation évènementielle

Que se passe-t-il ?

- Une interface mais pas de réaction ?
- Le système doit détecter ce qui survient…
  - ce qui a été fait
  - par quoi / qui ?
  - dans quel ordre ?
- …et doit adapter une réponse en avertissant les composants qui sont à l'écoute ou qui ont besoin d'être notifiés

- Pour cela, on se base sur la programmation évènementielle.

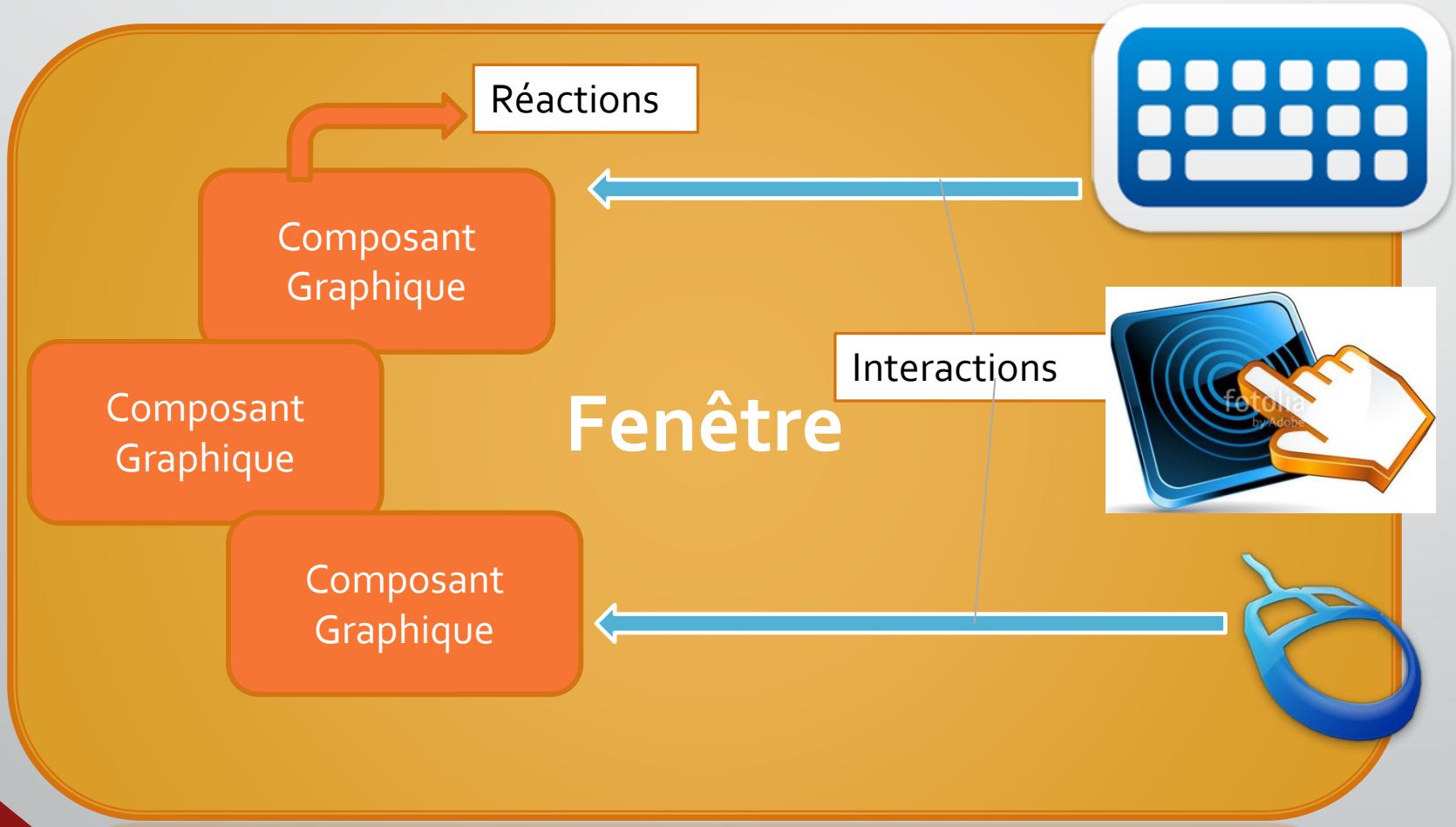
Programmation impérative

- Elle décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme
- Types d'instructions principales :
  - la séquence d'instructions,
  - l'assignation (ou affectation),
  - l'instruction conditionnelle,
  - la boucle,
  - les branchements
- https://fr.wikipedia.org/wiki/Programmation_impérative
- Dans la programmation impérative, ce sont les séquences d'instructions qui pilotent le programme

Programmation évènementielle

- C'est un paradigme de programmation basé sur les évènements
- Ce sont les évènements qui pilotent le programme : concept adapté pour la réalisation d'interfaces graphiques
- Notion de composants :
  - interagissant entre eux et avec l'environnement
  - communication via des évènements initiées par
    - l'utilisateur via des périphériques : clavier, souris, écran
    - le système : présence ou chargement d'un fichier, déclenchement à une heure précise, etc.
  - réactions dictées par des comportements

Schématique



Programmation évènementielle

- Une fois votre interface créée et initialisée, il y a un **processus** qui effectue une surveillance continuelle des évènements.
- Ce n'est pas réellement un processus, c'est ce qu'on appelle un **thread** : c'est à dire une tâche (ou une unité d'exécution) d'un programme, rattachée à un processus.

- Pour résumer : votre application doit surveiller constamment les interactions susceptibles de déclencher des événements :
  - les actions de l'utilisateur
  - les changements détectés par le système

- Ce thread de surveillance ne doit pas être re-développé à chaque fois, il est fournit par JavaFX : c'est le **JavaFX Application Thread**

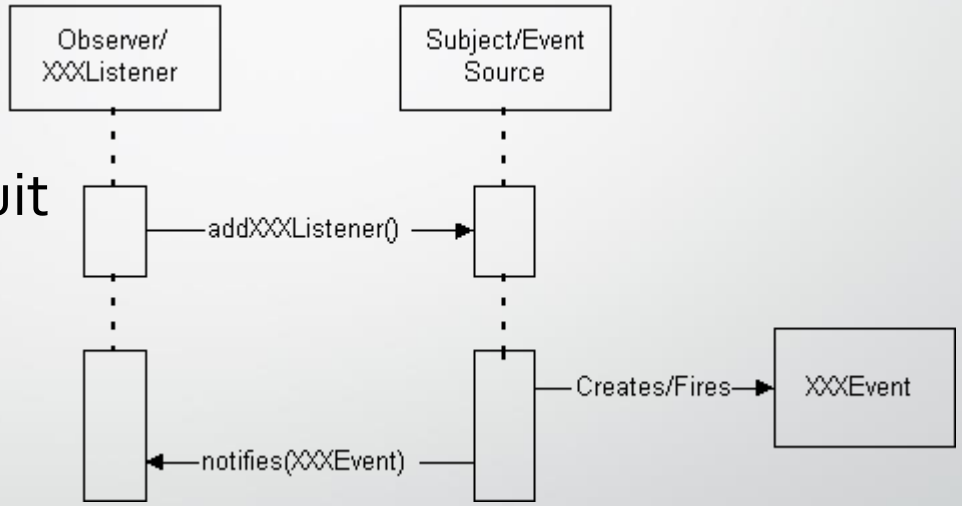
Exemple avec JavaFX

```
final Circle circle = new Circle(rayon, Color.RED);

circle.setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse entered");
    }
});

circle.setOnMouseExited(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse exited");
    }
});

circle.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse pressed");
    }
});
```

Le design pattern Observer

La délégation ?

- Le **sujet** enregistre un **observateur**
- Lorsqu'un **évènement** survient, le sujet signale qu'un évènement se produit
- L'observateur réagit à l'évènement auquel il est abonné, il est appelé
- **C'est donc à l'observateur qu'est délégué le traitement**



Le pattern Observer

Rappel : un pattern ?

- **Pattern ou Design Pattern (patron de conception) :**
  - Ce sont des solutions générales, réutilisables, que l'on peut appliquer à des problèmes informatiques
  - Ils sont dépendants d'un contexte spécifique et y répondent
  - Ils donnent des indications aux développeurs / concepteurs et définissent un haut niveau de bonnes pratiques

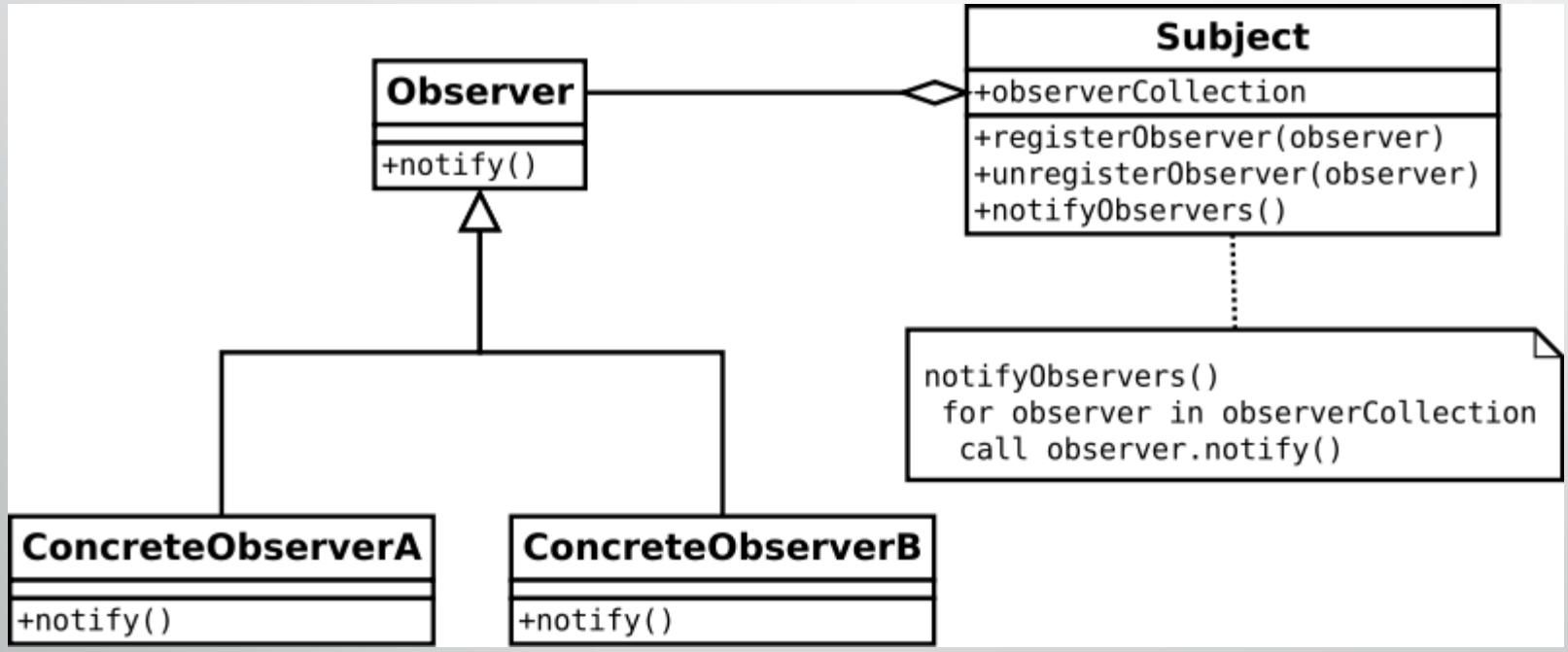
Rappel : un pattern ?

- **Les plus connus sont formalisés dans le livre intitulé Design Patterns – Elements of Reusable Object-Oriented Software publié en 1994 chez Addison-Wesley.**
- **Livre de méthodologie appliquée à la conception logicielle écrit par le « Gang of Four » (abrégé GoF) :**
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides

Familles de pattern

- **On distingue trois familles de patrons de conception selon leur utilisation :**
  - **de construction** : ils définissent comment faire l'instanciation et la configuration des classes et des objets
  - **structuraux** : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation)
  - **comportementaux** : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués

Le pattern Observer



Le pattern Observer

- Exemple en Java

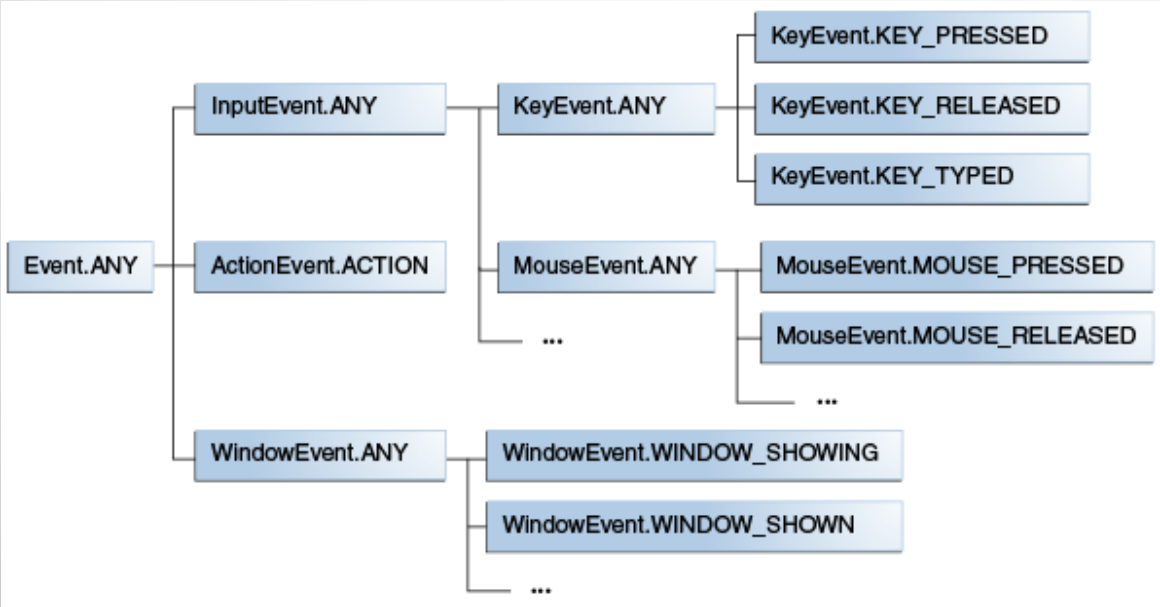
Les évènements dans JavaFX

- Héritent de la classe Event : ce sont des objets
- De nombreux évènements sont prédéfinis dans JavaFX
- On peut aussi créer des évènements personnalisés
- Chaque évènement contient entre autres :
  - **le type** : instance de EventType obtenu par *getEventType()*
  - **la source** : Object obtenue par *getSource()*
  - **la cible** : EventTarget obtenue par *getTarget()*

- Le **EventType** est lui-même un objet : il possède un nom (obtenu via *getName()*) et un type parent (*getSuperType()*)
- https://docs.oracle.com/javafx/2/events/jfxpub-events.htm

Types d'évènements 1/2

- Les types d'évènements sont classés de manière hiérarchique (héritage)
- Voici une partie de cette hiérarchie :



- A la racine, on a Event.ANY (équivalent à Event.ROOT)

https://docs.oracle.com/javafx/2/events/processing.htm

Types d'événements 2/2

- ActionEvent
- KeyEvent : onKey (pressed, typed, released)
- MouseEvent : onMouse (clicked, dragged, entered, exited, moved, pressed, released)
- ScrollEvent : onScroll (started, finished)
- ZoomEvent : onZoom (started, finished)
- RotateEvent : onRotation (started, finished)
- DragEvent : onDrag (over, dropped)
- SwipeEvent : onSwipe (down, up, right, left)
- TouchEvent : onTouch (moved, pressed, released, stationary)
- MediaErrorEvent
- ContextMenuEvent, EditEvent, CellEditEvent
- WindowEvent (affichée, fermée, masquée)
- WebEvent (alert, error, resized, statusChanged)

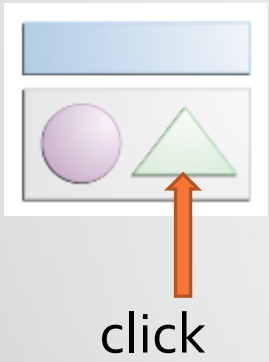
(liste non exhaustive)

La propagation des évènements 1/2

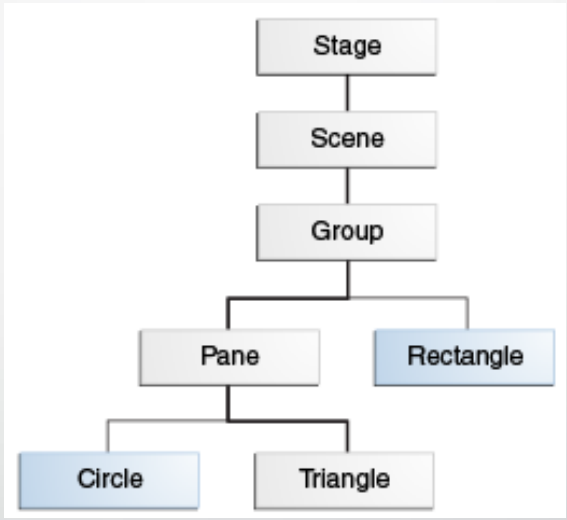
- Les évènements se propagent selon 4 étapes :
  - Sélection de la **cible** : le système détermine le nœud cible (ou target) de l'évènement
  - Construction de la **route** : la route parcourue par l'évènement (dans le graphe de scène) pour atteindre sa target, determinée par le "*event dispatch chain*"
  - Phase de "**capture**" : l'évènement se propage vers le bas, à partir du noeud racine (Stage) jusqu'à sa cible
  - Phase de "**bubbling**" : l'évènement se propage vers le haut, de la cible pour remonter vers la racine

La propagation des évènements 2/2

- Exemple : si un utilisateur clique sur le triangle, l'évènement suit la route des nœuds en gris



capture



bubbling

La route de propagation peut être modifiée (non traité dans ce cours).

Si un nœud consomme l'évènement (méthode ***consume***), les nœuds suivants (sur la route) ne seront pas notifiés.

Principe des écouteurs

- Pour que notre interface soit notifiée quand un évènement survient, il faut :
  - créer un « **écouteur** » (ou **Listener**) pour cet évènement
  - enregistrer cet écouteur auprès d'un nœud du graphe
  - l'écouteur est ensuite « réveillé » par le **JavaFX Application Thread** dès l'instant où l'évènement qu'il écoute se produit sur le nœud concerné
  - chaque Listener définit les actions à entreprendre, dans des méthodes prévues à cet effet (interface prédéfinie : *handle*)
- Il existe 2 sortes d'écouteurs : les **Event Filters** et les **Event Handlers** (toutes deux implémentent l'interface *EventHandler*)

Les "Event Filters"

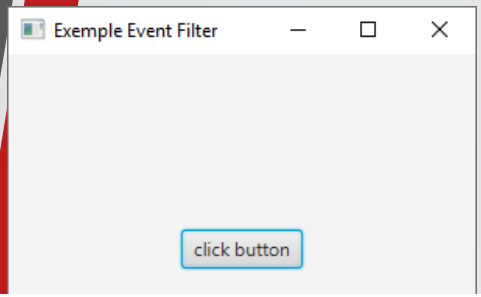
- S'exécutent lors de la phase descendante dite de capture
- Un nœud peut avoir un ou plusieurs filtres enregistrés
- Un même filtre peut être utilisé sur différents nœuds, et peut être appliqué à différents types d'évènements
- Un filtre peut consommer l'évènement et stopper sa propagation : ***event.consume( )***
- Méthodes pour enregistrer ou supprimer un filtre d'un nœud :

addEventFilter(EventType<T> eventType, EventHandler<? super T> eventFilter)

removeEventFilter(EventType<T> type, EventHandler<? super T> filter)

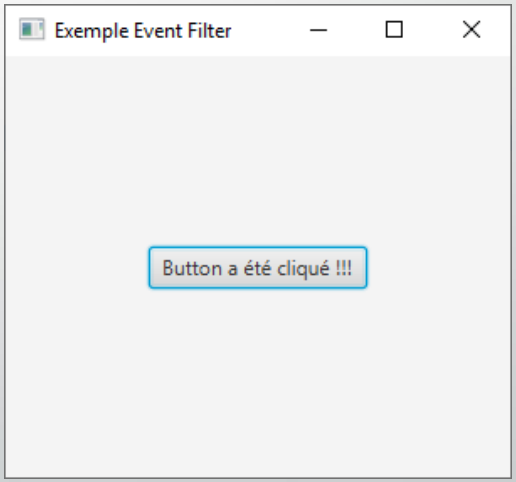
- On choisit le mode d'instanciation du filtre (3 possibles)
- Dans la méthode callback de l'écouteur on détermine les actions à mener : ***void handle(T event)***

Un filtre simple




```java
@Override
public void start(Stage primaryStage) {

    Button btn1 = new Button();
    btn1.setText("click button");

    // Création d'un Layout Pane de type HBox
    HBox root = new HBox();
    root.setAlignment(Pos.CENTER);
    root.getChildren().add(btn1);

    Scene scene = new Scene(root, 300, 250);

    primaryStage.setTitle("Exemple Event Filter");
    primaryStage.setScene(scene);
    primaryStage.show();
}

/**
 * @param args arguments passés à la JVM
 */
public static void main(String[] args) {
    Application.launch(args);

}
```

```java
btn1.addEventFilter(MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {

    @Override
    public void handle(MouseEvent event) {
        btn1.setText("Button a été cliqué !!!");
    }

});
```



démonstration

Que s'est-il passé ?

- L'utilisateur a cliqué sur le composant Button,
- Le Button a averti son/ses écouteur(s), et leur a transmis l'ensemble des informations utiles (xxEvent, ici MouseEvent).
- L'évènement propagé est récupéré par le/les écouteur(s).
- Le code spécifique à chaque écouteur est exécuté (en s'appuyant ou pas sur xxEvent)

Écouteurs : comment faire ?

- Il y a trois manières de déclarer un écouteur en JavaFX :
  - une classe anonyme (cf. exemple précédent)
  - une classe interne (ou classe imbriquée)
  - une classe externe (joue le rôle de contrôleur du composant)

La classe anonyme

- C'est une classe locale, mais non nommée
  - pas de déclaration dans le code = pas de variable référencée
  - concision du code
  - utilisation unique
- Elle peut :
  - accéder aux membres de la classe qui la détient
  - masquer des déclarations
  - redéfinir des opérations
- Elle ne peut pas :
  - avoir d'initialisateur static

A privilégier pour des utilisations uniques, simples, sans ré-utilisation. Attention à ne pas faire du code "poubelle" en multipliant les classes anonymes !

La classe interne

- C'est une classe déclarée à part entière
  - Elle n'existe que dans la classe qui l'inclut
  - Elle accède aux membres de la classe l'incluant
  - Elle permet de compléter le fonctionnement de la classe l'incluant
  - Intéressant pour de multiples instanciations / ré-utilisation
  - Permet de clarifier le code utilisé

La classe interne : exemple

```
public class HelloWorld extends Application {

    @Override
    public void start(Stage primaryStage) {

        Button btn1 = new Button();
        btn1.setText("click button");

        // ajout d'un Event Filter via une classe interne
        btn1.addEventFilter(MouseEvent.MOUSE_PRESSED, new FilterInterne());

        HBox root = new HBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(btn1);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Exemple Event Filter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    class FilterInterne implements EventHandler<MouseEvent>{

        @Override
        public void handle(MouseEvent event) {
            System.out.println("Button a été cliqué !!!");
        }

    }
```

La classe externe : exemple

```java
public class HelloWorld extends Application {

    @Override
    public void start(Stage primaryStage) {

        Button btn1 = new Button();
        btn1.setText("click button");

        // ajout d'un Event Filter via une classe interne
        btn1.addEventFilter(MouseEvent.MOUSE_PRESSED, new ButtonController(btn1));

        HBox root = new HBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(btn1);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Exemple Event Filter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

```java
public class ButtonController implements EventHandler<MouseEvent> {

    private Button btn;

    // constructeur qui prend en argument
    // des références des composants à modifier
    public ButtonController(Button btn) {
        this.btn = btn;
    }

    @Override
    public void handle(MouseEvent event) {
        // on modifie le libellé du composant
        btn.setText("Button a été cliqué !!!");
    }

}
```

Les "Event Handlers"

- C'est un autre type d'écouteur d'évènements
- La principale différence avec les filtres réside dans le fait qu'ils s'exécutent lors de la phase montante dite de bubbling

  (donc après les Event Filters)

- Pour le reste, c'est le même fonctionnement que les filtres

- Méthodes pour enregistrer/supprimer un handler d'un nœud :

addEventHandler(EventType<T> type, EventHandler<? super T> handler)

removeEventHandler(EventType<T> type, EventHandler<? super T> handler)

Les "convenience methods"

- Certaines classes JavaFX possèdent un attribut de type EventHandler.
- Le setter permet alors d'enregistrer facilement un handler auprès de cette classe
- C'est ce que nous appelons une "convenience method" (ou méthode utilitaire)

  - *Format :*

    setOn*EventType(EventHandler<? super event-class> value)*

  - *Exemple :*

    setOn*KeyTyped(EventHandler<? super KeyEvent> value)*

https://docs.oracle.com/javafx/2/events/convenience_methods.htm

Action utilisateur	Type évènement	Classes
Une touche du clavier est pressée	KeyEvent	Node, Scene
La souris est déplacée ou cliquée	MouseEvent	Node, Scene
Glisser-déposer avec la souris (Drag-and-Drop)	MouseDragEvent	Node, Scene
Saisie de texte modifiée	InputMethodEvent	Node, Scene
Glisser-déposer spécifique à la plateforme	DragEvent	Node, Scene
Défilement (scroll)	ScrollEvent	Node, Scene
Rotation	RotateEvent	Node, Scene
Balayage / défilement (swipe)	SwipeEvent	Node, Scene
Touché (tactile)	TouchEvent	Node, Scene
Geste de zoom (gesture)	ZoomEvent	Node, Scene

https://docs.oracle.com/javafx/2/events/convenience_methods.htm		
Action utilisateur	**Type évènement**	**Classes**
Ouverture d'un menu contextuel	ContextMenuEvent	Node, Scene
Appui sur un bouton, ComboBox ouverte ou fermée, Choix d'un menu	ActionEvent	ButtonBase, ComboBoxBase, ContextMenu, MenuItem, TextField
Édition d'un élément d'une liste, d'une table ou d'un arbre (tree)	ListView.EditEvent TableColumn.CellEditEvent TreeView.EditEvent	ListView TableColumn TreeView
Erreur survenue dans le Media Player	MediaErrorEvent	MediaView
Menu affiché ou masqué	Event	Menu
Fenêtre pop-up masquée	Event	PopupWindow
Un onglet (Tab) est sélectionné ou fermé	Event	Tab
Fenêtre est fermée, affichée ou masquée	WindowEvent	Window