Les évènements en javaFX

28 janvier 2021

Introduction

Programmation évènementielle :

- Autre paradigme de programmation que la programmation séquentielle.
  - "Évènements" gérés par le système,
  - on écrit les "réacteurs" aux évènements (qui peuvent générer d'autres évènements).
- Bien adapté (mais pas que) à la programmation des interfaces :
  - Mathématiques : *Mathlab*
  - Réseau : *Node.js*, *Twisted*
  - Web : *Nginx*
  - et IHM : `TCL/TK`, *AWT/Swing*, `Qt`, `Windows API`, etc...

Introduction: Principe général.

Fonctionnement :

- Event : il y a des **évènements** :
  – évènements prédéfinis : clic souris, saisie de texte, terminaison d'un thread...
  – ou nos propres classes d'évènements, lancés par nous même.
- EventHandler : on écrit des **écouteurs** chargés d'agir lors d'un évènement :
  – l'*écouteur* ou *EventHandler* est l'objet à l'écoute,
  – le *handler* est la méthode de l'écouteur exécutée lors de l'évènement.
  Remarque : l'exécution du handler peut causer d'autres évènements.
- Abonnement : on **abonne** (ou on enregistre) l'écouteur à :
  – un type d'évènement,
  – sur un composant particulier.
- JavaFX Application Thread gère (entre autre) les évènements.
  - traite les interactions du système, propage si besoin les évènements, et les met dans une file d'attente de traitement.
  - pour chaque évènement sur un composant :
    il exécute le code des écouteurs qui y ont été abonnés.

Introduction: Principe général.

Event

Il y a des évènements :

- de bas niveau : interaction directe avec le windows system
  - action sur le bouton de la souris : enfoncement, relachement, clic.
  - mouvement de la souris : arrivée sur le composant, drag and drop, ...
  - gestion clavier : enfoncement, relachement, production d'un caractère.
  ...
- de haut niveau : traités par javaFX
  - action d'un bouton (peut importe la façon de l'actionner : clic souris, toucher écran, raccourci clavier).
  - changement de texte (peut importe la façon : clavier, copié à la souris, etc...)
  ...
- "foreground" : résultat de l'action directe de l'utilisateur.
- "background": indépendant de l'action directe de l'utilisateur ; processus qui se termine, interruption système, connection réseau, etc..

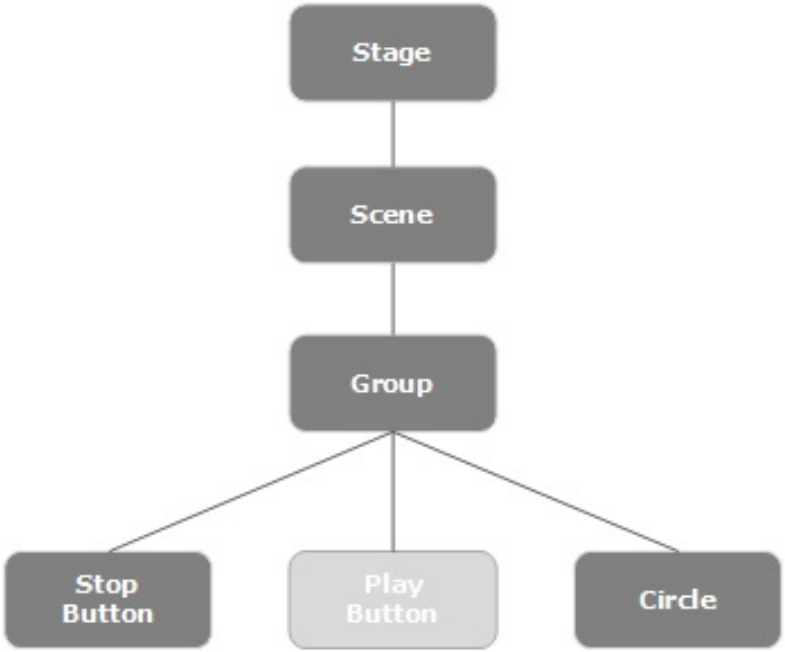
Introduction: Principe général.

Le JavaFX Application thread traite :
– la construction de l'interface,
– la gestion des évènements,
– l'exécution des handlers.

• Lorsque des évènements arrivent, ce thread les traite dans l'ordre d'arrivée, *"le suivant* **après avoir fini** *le précédent".*

⇒ l'interface est bloquée tant que le traitement n'est pas fini,
⇒ un *handler* doit s'exécuter le plus rapidement posssible,
⇒ déporter les longs calculs dans un autre thread.

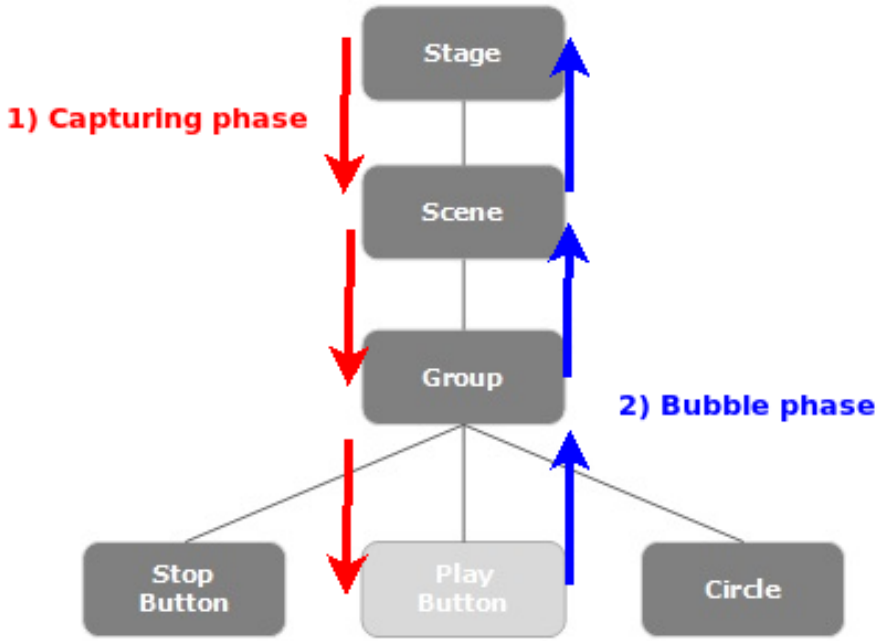
Introduction: la distribution dans la scène



Que se passe-t-il lors d'une action sur le bouton play ?

Introduction: la distribution dans la scène

L'évènement suit la Event Dispatch Chain :



1. capture : l'évènement est propagé de la frame jusqu'au bouton (la cible),
2. remontée : l'évènement est propagé du bouton jusqu'à la frame.

Introduction: la distribution dans la scène

L'évènement peut-être traité sur chacun des noeuds de l'E.D.C., à la descente comme à la remontée.
Il suffit d'abonner un EventHandler à cet évènement sur ce noeud.
– Filter (filtre) : EventHandler actionné lors de la phase de capture (à la descente).
– Handler (gestionnaire) : EventHandler actionné lors de la phase de bubbling (à la remontée).

- $\Rightarrow$ deux méthodes que possèdent tous les noeuds : `addEventFilter` pour abonner un filtre, `addEventHandler` pour abonner un gestionnaire.
- on peut arrêter la propagation de (ou consommer) l'`Event e` en appelant `e.consume()`.
- **dans la très grande majorité des cas, l'évènement est traité dans un `Handler` au niveau de la cible.**

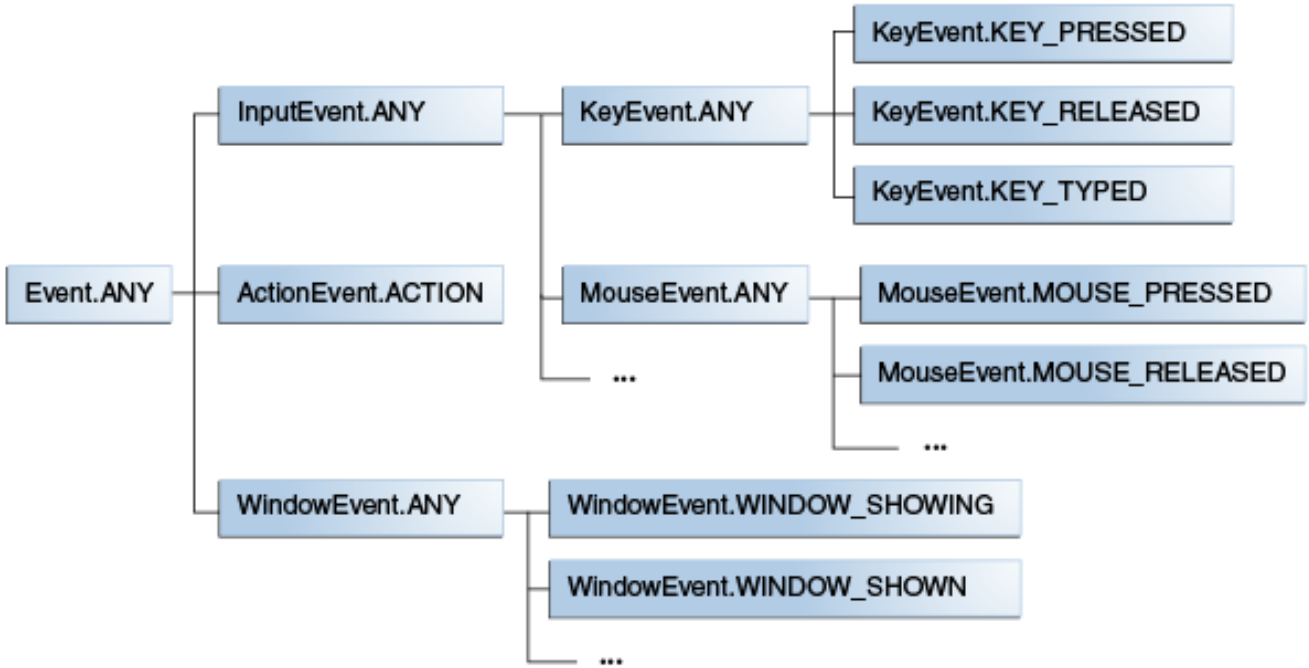
Propriétés des évènements

Tout évènement a :

- un *Type* : `eventType getEventType()`
  définit ce qu'il s'est passé,
  (`ActionEvent.ACTION`, `KeyEvent.KEY_PRESSED`,
  `MouseEvent.MOUSE_CLICKED`...) ;
- une *cible* ou *Target* : `getTarget()`
  composant cible de l'évènement (le bouton dans l'exemple précédent),
- une *Source* : `Object getSource()`
  composant ayant déclenché le `EventHandler`.
- **suivant sa classe**, des attributs et méthodes particulières.
  Par exemple `getX()` et `getY()` pour `MOUSE_EVENT`

Évènements : Hierarchie des EventType

- Chaque type d'évènement a un type parent.



Extrait de la hiérarchie des types d'évènement

- *Reflète* la hierarchie des classes `Event`.
- Toute classe a un évènement parent `ANY`.
  Cela permet de capturer tous les types d'évènements de la classe. Par exemple tous les évènements de souris, que ce soit un clic, un mouvement, un "drag and drop"

Évènements : Event

`Event`

- eventType : `Event.ANY`
- méthodes particulières : aucune

Racine de la hierarchie des Event.

$\Rightarrow$ si sur un noeud, on pose un handler réagissant aux `Event.ANY`, ce Handler va réagir à tous ce qui passe par ce noeud : `ActionEvent.ACTION`, `MouseEvent.MOUSE_MOVED`, `KeyEvent.Key_PRESSED`, etc...

Évènements : ActionEvent

`ActionEvent`

- eventType : `ActionEvent.ANY` `ActionEvent.ACTION`
- méthodes particulières : aucune

Évènements : MouseEvent

`MouseEvent`

- eventType : `MouseEvent.ANY`
  `MouseEvent.CLICKED`,`MouseEvent.PRESSED`,`MouseEvent.RELEASED`,`MouseEv`
- méthodes particulières pour :
  - la géométrie : `getX` `getY`, `getZ`, `getSceneX`... , `getScreenX`,...
  - le clic et l'état de la souris : `getButton`, `getCLickCount`, `isPrimaryButtonDown`,...
  - les modifieurs : `isAltDown`, `isControlDown`,...
  - ...

Attention : l'action de la molette est dans `ScrollEvent`

Évènements : InputMethodEvent

`InputMethodEvent`

concerne essentiellement les saisies/modifications de texte.

- eventType :
  `InputMethodEvent.ANY`,
  `InputMethodEvent.INPUT_METHOD_TEXT_CHANGED`
- méthodes particulières :
  - la position du curseur : `int getCaretPosition()`
  - le texte : `String getCommited`, `getComposed`,

Évènements : KeyEvent

`KeyEvent`

action sur le clavier.

- eventType : `ANY`, `CHAR_UNDEFINED`, `KEY_PRESSED`, `KEY_RELEASED`, `KEY_RELEASED`
- méthodes particulières pour :
  - la touche :
    - `String getCharacter()` (en Unicode),
    - `KeyCode getCode()`,
    - `String getText`
  - les modifieurs : `isAltDown`, `isControlDown`,...

Évènements : WindowEvent

`WindowEvent`.

- eventType : `ANY`, `WINDOW_CLOSE_REQUEST`, `WINDOW_HIDING`, `WINDOW_HIDDEN`, `WINDOW_SWHOWING`, `WINDOW_SHOWN`
- pas de méthode particulière.

Évènements : Autres Event

D'autres classes d'évènements : chacune spécialisée pour un composant ou une interaction particulière :

`DialogEvent`, `TouchEvent`, `ContextMenuEvent`, `WebEvent`...

EventHandlers

EventHandler Je vais dire souvent *écouteurs* (héritent de la classe `Listener`).

Il y plusieurs manières de programmer (et d'abonner) des EventHandlers. Fondamentalement il se passe :

1. écriture d'une classe `Ecouteur` implémentant `EventHandler`
2. création d'une instance `monEcouteur` de cette classe
3. abonnement de :
   - l'Ecouteur `monEcouteur`,
   - sur un composant `comp`,
   - pour un type d'évènement `typeEv`,

   avec `comp.addEventHandler(typeEv,monEcouteur)`
   et/ou `comp.addEventFilter(typeEv,monEcouteur))`

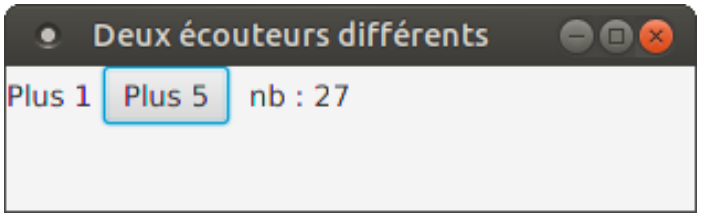
On verra ensuite les `raccourcis` qu'offre Java.

EventHandlers

public interface **EventHandler**<T extends Event>

- C'est une interface $\Rightarrow$ *n'importe quel objet peut être un écouteur*
- Une unique méthode abstraite : `void handle(T event)`
  - C'est la méthode qui est appelée lorsque l'écouteur est "réveillé" par un type d'évènement sur lequel il est abonné.
  - **on écrit dans `handle` ce qu'on veut que l'écouteur fasse**.
  - `event` est l'évènement qui a réveillé l'écouteur.
    Exemple : Si `event` est un `MouseEvent`, `event.getX()` donne la position en X du clic.

EventHandlers : Exemple



- `Label lCompteur` : *"nb : ..."*
- `Label lPlus1` : *"Plus 1"*
  le survol par la souris fait +1 sur le Label *"nb : ..."*
- `Button bPlus5` : *"Plus 5"*
  l'action sur le bouton fait + 5 sur le Label *"nb : ..."*
- int compteur : compte les clicks.

EventHandlers : un écouteur par composant

1. écriture de la classe Ecouteur (en InnerClass)

La méthode `handle` incrémente `compteur` et change le texte de `lCompteur`

```
public class Ecouteur implements EventHandler{
  int increment ;

  public Ecouteur(int increment){
    this.increment = increment ;
  }

  @Override
  public void handle(Event event) {
    compteur = compteur+increment ;
    lCompteur.setText(" nb : "+compteur);
  }
}
```

EventHandlers : un écouteur par composant

2. instantiation et abonnement des deux écouteurs

```
Ecouteur ePlus1 = new Ecouteur(1);
Ecouteur ePlus5 = new Ecouteur(5);

lPlus1.addEventHandler(MouseEvent.MOUSE_ENTERED, ePlus1);
bPlus5.addEventHandler(ActionEvent.ACTION, ePlus5);
```

EventHandlers : écouteur utilisant des méthodes de Event

La méthode `handle` incrémente `compteur` et change le texte de `lCompteur`, elle incrémente de 1 ou 5, en testant quel est le composant cible.

```
public class Ecouteur implements EventHandler{

  @Override
  public void handle(Event event) {
    if (event.getSource() == lPlus1)
      compteur++ ;
    else
      compteur = compteur+5 ;
    lCompteur.setText(" nb : "+compteur);
  }
}
```

EventHandlers : écouteur utilisant des méthodes de Event

2. instantiation et abonnement de l'écouteur

```
Ecouteur ePlus = new Ecouteur();

lPlus1.addEventHandler(MouseEvent.MOUSE_ENTERED, ePlus);
bPlus5.addEventHandler(ActionEvent.ACTION, ePlus);
```

EventHandlers : écouteur qui est la classe elle

La classe elle même où se situe le composant peut être l'écouteur.
Imaginons que le composant *b* écouté soit dans l'Application `MonApp`

1. `public class monApp extends Application` **implements EventHandler** {
2. on écrit la méthode `handle` **dans** la classe CompteurStage1E.

   ```
   public void handle(Event e) {...
   ...
   }
   ```

3. on abonne en faisant `b.addEventHandler(eventType,` **this**`)`

EventHandlers : ordre d'exécution

• Sur un même composant, on peut abonner plusieurs écouteurs différents, écouteurs qui peuvent réagir à la même interaction.
Exemple : un clic de souris sur un bouton peut déclencher un `EventAction.ACTION`, et un `MouseEvent.MOUSE_CLICKED`

• Soient les abonnements :
- `c.addEventHandler(eventType1, ecouteur1)`
- `c.addEventHandler(eventType2n, ecouteur2)`
(ou `addEventFilter(..)`)

Questions : dans quel ordre sont-ils exécutés ?
Que se passe-t-il si un des deux écouteurs consomme l'évènement ?

EventHandlers : ordre d'exécution

Ordre d'exécution :

- comme on l'a déjà dit, les *filtres* avant les *gestionnaires*
- si `eventType1` et `eventType2` ne sont pas descendants l'un de l'autre dans la hiérarchie des `eventTypes`, l'ordre d'execution n'est pas spécifié.
- si `eventType1` est descendant de (i.e. plus spécifique que) `eventType2` , alors `ecouteur1` est exécuté en premier.
- la méthode`e.consume()` n'interrompt pas le traitement des autres écouteurs du même noeud.

EventHandlers : abonnement avec setOn*xxx*

La plus part du temps, on abonne un écouteur :

- directement sur le composant **cible** de l'évènement,
- comme **gestionnaire** (juste au début de la "bubble" phase)
- pour un **eventType que peut avoir** la *cible*, par exemple :
  - un ActionEvent.ACTION, sur un `Button`, ou un `TextField`
  - un KeyEvent.KeyPressed sur un `TextField`,
  - un MouseEvent.MOUSE_ENTERED, sur un noeud.

  Et un noeud ne peut être cible que de certains évènements.
- on abonne un seul évènement par noeud

EventHandlers : abonnement avec setOn*xxx*

$\Rightarrow$ Méthodes spécialisées pour ce type d'abonnement.

• Pour (presque ?) chaque type d'évènement *xxx* que peut créer une classe `C`, la classe `C` contient une méthode `setOn`*xxx*.

Exemples :

- `button.setOnAction(EventHandler<ActionEvent> e)`
- `node.setOnMouseMoved(EventHandler<? super MouseEvent> e)`
- `textField.setOnKeyPressed(...)`

EventHandlers : abonnement avec setOn*xxx*

Liste (extrait) des types d'évènements que peut lancer chaque classe de noeud : chercher les méthodes `setOn`*`eventType`* dans la classe.

Classe	Action de l'utilisateur	Évènement
Node, **Scene**	Pression sur une touche clavier	KeyEvent
	mouvement/pression souris	MouseEvent
	glisser/déposer souris	MouseDragEvent
	composant touché	TouchEvent
	geste de zoom	ZoomEvent
	composant scrollé	ScrollEvent
	texte modifié (durant la saisie)	InputMethodEvent
	activation menu contextuel	ContextMenuEvent
ButtonBase	bouton cliqué	
ComboBoxBase	combobox ouverte ou fermée	
ContextMenu	une des option d'un menu activée	ActionEvent
MenuItem	option de menu activée	
TextField	pression sur *Enter* dans un champ texte	

EventHandlers : abonnement avec setOn*xxx*

Classe	**Action de l'utilisateur**	`Évènement`
Menu	menu déroulé, ou enroulé	Event
PopupWindow	fenetre *popup* masquée	
Tab	onglet sélectionné ou masqué	
Window	fenêtre affichée, fermée, masquée	WindowEvent
...	...	...

"Raccourcis" java

Très souvent :

- pour un certain évènement sur un certain composant, l'écouteur est utilisé une seule fois (inutile donc d'écrire une classe Ecouteur réutilisable).
- le code de la méthode handler est très court.

On peut utiliser les "raccourcis" java pour faire la création de l'EventHandler lors de l'abonnement en une seule opération.

"Raccourcis" java : avec EventHandler local anonyme

C'est la méthode proposée dans le `HelloWorld.java`

```
lPlus1.setOnMouseEntered(new EventHandler(){
        @Override
        public void handle(Event event) {
            compteur++ ;
            lCompteur.setText(" nb : "+compteur);
        }
```

- On crée un objet `EventHandler` anonyme, à qui on passe la méthode handle souhaitée.
- Peut-être fait avec `addEventHandler()` ou `addEventFilter()`.

"Raccourcis" java : avec expression lambda

```
bPlus1.setOnAction(
            event -> {
                compteur = compteur ++;
                lCompteur.setText(" nb : " + compteur);
            }
            );
```

- *java* utilise :
  - le profil de *setOnAction* pour voir qu'il faut un EventHandler.
  - la lambda expression *(un paramètre de type non précisé)-> (un bloc de code)*
  - pour trouver la méthode de `EventHandler` qui correspond (avec un seul parametre, peut importe son type). Cela tombe bien, il n'y en a qu'une et elle correspond.

  Il écrit `handler(Event event)` avec le bloc de code, instancie l'éventHandler et le passe en paramètre.
- Peut-être fait avec `addEventHandler()` ou `addEventFilter()`.