Cours IHM-1
JavaFX

6 - Architecture MVC
Gestion des événements

Architecture MVC
Structure d'une application

Architecture MVC [1]

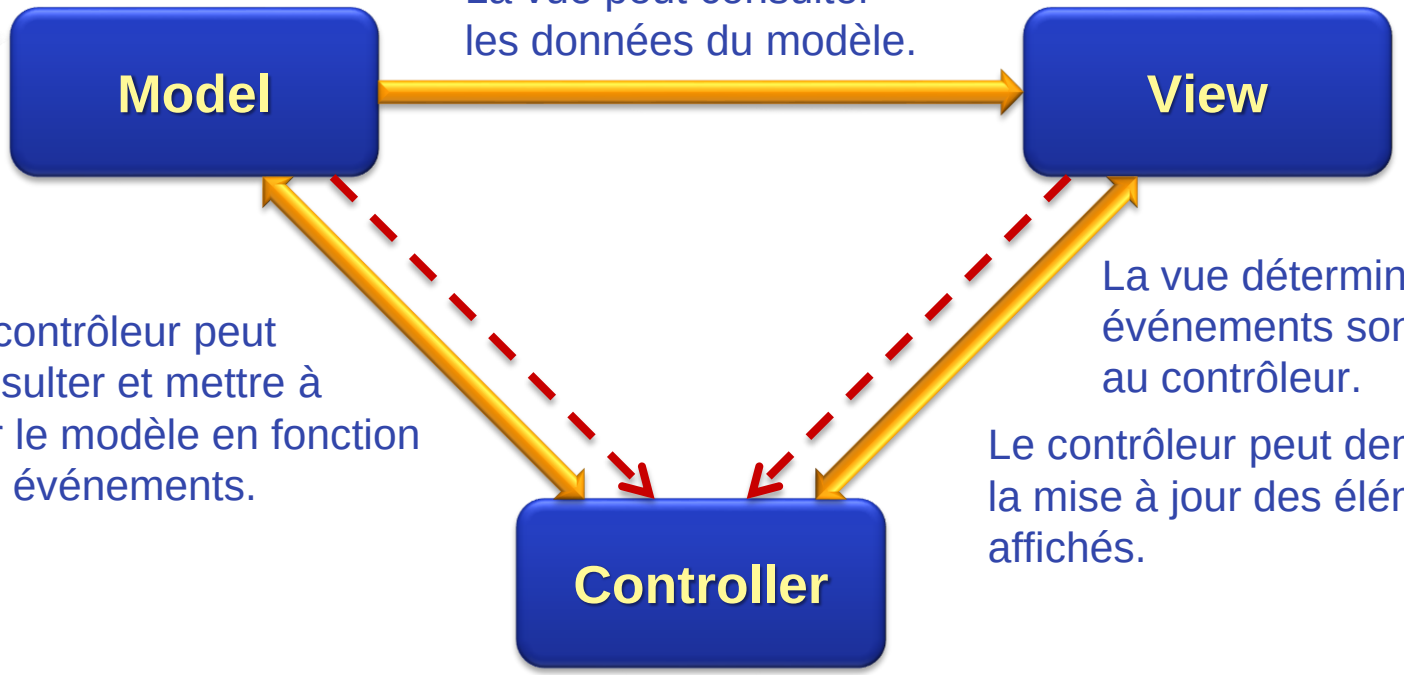
- Il existe différentes manières de structurer le code des applications interactives (celles qui comportent une interface utilisateur).
- Une des architectures, communément utilisée, et qui comporte de nombreuses variantes, est connue sous l'acronyme **MVC** qui signifie **M**odel - **V**iew - **C**ontroller.
- Dans cette architecture on divise le code des applications en entités distinctes (*modèles*, *vues* et *contrôleurs*) qui communiquent entre-elles au moyen de divers mécanismes (invocation de méthodes, génération et réception d'événements, etc.).
- Cette architecture (ou modèle de conception, *design pattern*) a été introduite avec le langage *Smalltalk-80* dans le but de simplifier le développement ainsi que la maintenance des applications, en répartissant et en découplant les activités dans différents sous-systèmes (plus ou moins) indépendants.

Architecture MVC [2]

- Le principe de base de l'architecture MVC est relativement simple, on divise le code du système interactif en trois parties distinctes :
  - Le ou les **modèles** ***(Models)*** qui se chargent de la gestion des données (accès, transformations, calculs, etc.). Le modèle enregistre (directement ou indirectement) l'état du système et le tient à jour.
  - Les **vues** ***(Views)*** qui comprennent tout ce qui touche à l'interface utilisateur (composants, fenêtres, boîtes de dialogue) et qui a pour tâche de présenter les informations (visualisation). Les vues participent aussi à la détection de certaines actions de l'utilisateur (clic sur un bouton, déplacement d'un curseur, geste *swipe*, saisie d'un texte, …).
  - Les **contrôleurs** ***(Controllers)*** qui sont chargés de réagir aux actions de l'utilisateur (clavier, souris, gestes) et à d'autres événements internes (activités en tâches de fond, *timer*) et externes (réseau, serveur).
- Une application peut également comporter du code qui n'est pas directement affecté à l'une de ces trois parties (librairies générales, classes utilitaires, etc.).

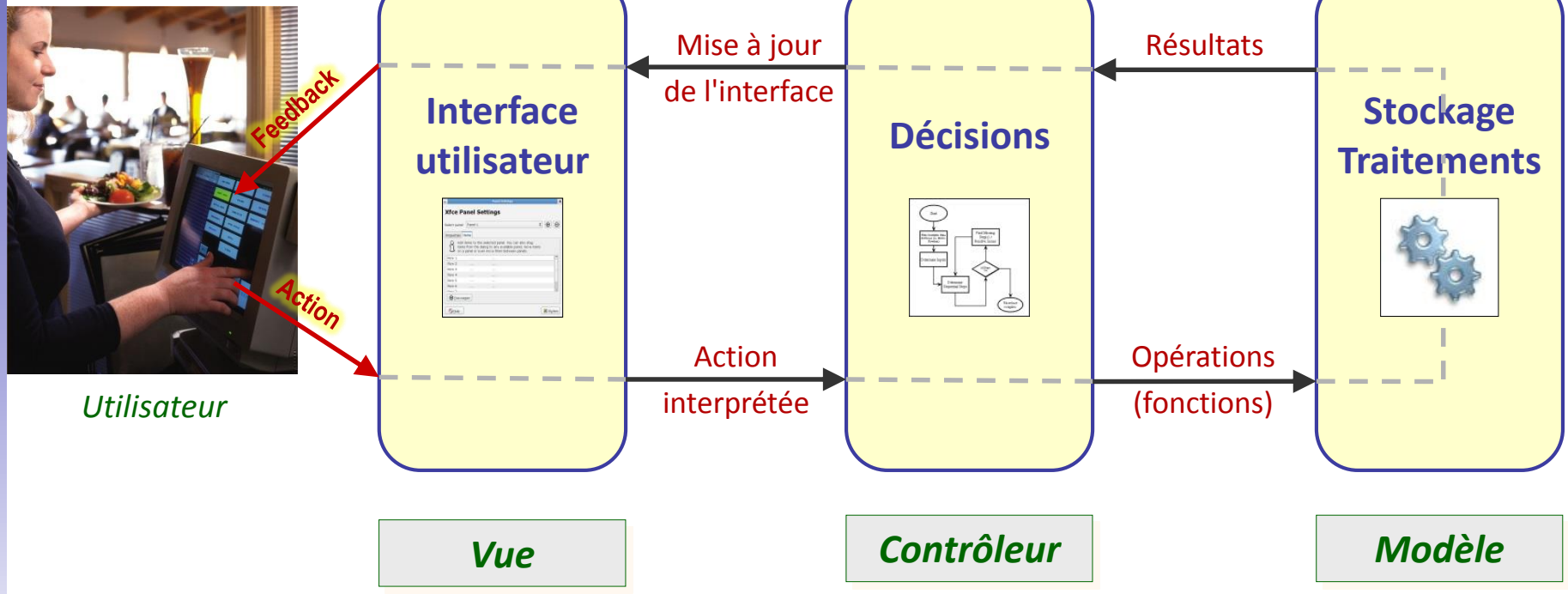
Interactions MVC [1]

- Un exemple de communication entre les éléments (MVC synchrone).



Interactions MVC [1]

- Lorsqu'un utilisateur interagit avec une interface, les différents éléments de l'architecture MVC interviennent pour interpréter et traiter l'événement.



MVC – Modèle

- Le **modèle** ***(Model)*** est responsable de la gestion des données qui caractérisent l'état du système et son évolution.
- Dans certaines situations (simples) le modèle peut contenir lui-même les données mais, la plupart du temps, il agit comme un intermédiaire *(proxy)* et gère l'accès aux données qui sont stockées dans une base de données, un serveur d'informations, le *cloud*, …
- Le modèle est souvent défini par une ou plusieurs interfaces *Java* qui permettent de s'abstraire de la façon dont les données (les objets *métier*) sont réellement stockées (notion de DAO *Data Access Object*).
- Il offre également les méthodes et fonctions permettant de gérer, transformer et manipuler ces données.
- Les informations gérées par le modèle doivent être indépendantes de la manière dont elles seront affichées. Le modèle doit pouvoir exister **indépendamment de la représentation visuelle** des données.

MVC – Vue

- La **vue** ***(View)*** est chargée de la représentation visuelle des informations en faisant appel à des écrans, des fenêtres, des composants, des conteneurs *(layout)*, des boîtes de dialogue, etc.
- Plusieurs vues différentes peuvent être basées sur le même modèle (plusieurs représentations possibles d'un même jeu de données).
- La vue intercepte certaines actions de l'utilisateur et les transmet au contrôleur pour qu'il les traite (souris, clavier, gestes, …).
- La mise à jour de la vue peut être déclenchée par un contrôleur ou par un événement signalant un changement intervenu dans les données du modèle par exemple (mode asynchrone).
- La représentation visuelle des informations affichées peut dépendre du *Look-and-Feel* adopté (ou imposé) et peut varier d'une plateforme à l'autre. L'utilisateur peut parfois modifier lui même le thème de présentation des informations.

MVC – Contrôleur

- Le **contrôleur** ***(Controller)*** est chargé de réagir aux différentes actions de l'utilisateur ou à d'autres événements qui peuvent survenir.
- Le contrôleur définit le comportement de l'application et sa logique (comment elle réagit aux sollicitations, *business logic*).
- Dans les applications simples, le contrôleur gère la synchronisation entre la vue et le modèle (rôle de chef d'orchestre).
- Le contrôleur est informé des événements qui doivent être traités et sait d'où ils proviennent.
- La plupart des actions étant interceptées (ou en lien) avec la vue, il existe un couplage assez fort entre la vue et le contrôleur.
- Le contrôleur communique généralement avec le modèle et avec la vue. C'est le sens des transferts et le mode de communication qui caractérisent différentes variantes de l'architecture MVC.

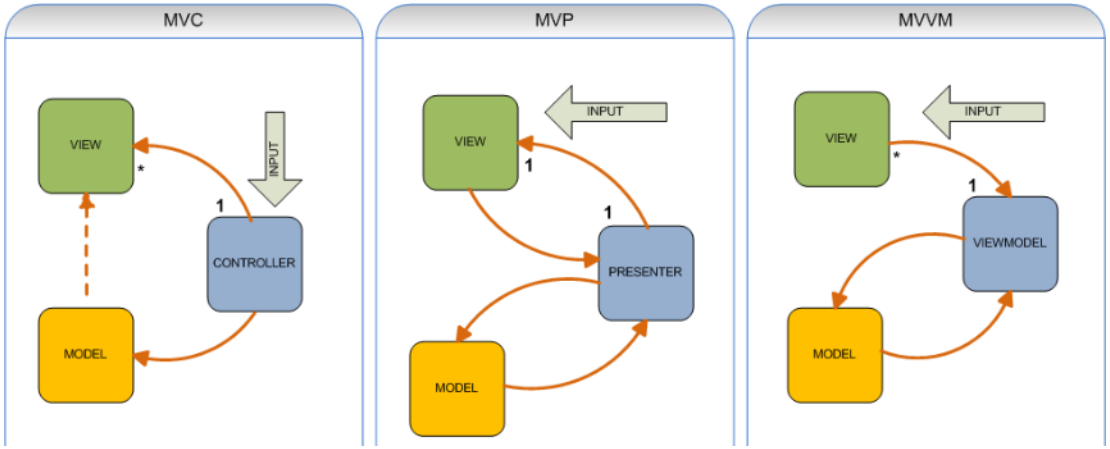
Structure d'une application [1]

- Une application *JavaFX* qui respecte l'architecture MVC comprendra généralement différentes classes et ressources :
  - Le **modèle** sera fréquemment représenté par une ou plusieurs classes qui implémentent généralement une interface permettant de s'abstraire des techniques de stockage des données.
  - Les **vues** seront soit codées en *Java* ou déclarées en FXML. Des feuilles de styles CSS pourront également être définies pour décrire le rendu.
  - Les **contrôleurs** pourront prendre différentes formes :
    - ⇨ Ils peuvent être représentés par des classes qui traitent chacune un événement particulier ou qui traitent plusieurs événements en relation (menu ou groupe de boutons par exemple)
    - ⇨ Si le code est très court, ils peuvent parfois être inclus dans les vues, sous forme de classes locales anonymes ou d'expressions lambda.
  - La **classe principale** (celle qui comprend la méthode `main()`) peut faire l'objet d'une classe séparée ou être intégrée à la classe de la fenêtre principale (vue principale).
  - D'autres **classes utilitaires** peuvent venir compléter l'application.

Variantes de l'architecture MVC [1]

- Il existe de nombreuses déclinaisons de l'architecture MVC ainsi que des variantes dont les plus connues sont :
  - **MVP** : *Model - View - Presenter*
  - **MVVM** : *Model - View - View-Model*
- Dans ces variantes le modèle et la vue sont définis de manière quasi identique. C'est le rôle du contrôleur et sa manière de communiquer avec les autres parties qui distinguent ces variantes de l'architecture MVC standard.

Ces différentes variantes d'architecture ne pourront pas être explorées plus en détail dans le cadre de ce cours.



Source: tomyrhymond.wordpress.com/2011/09/16/mvc-mvp-and-mvvm

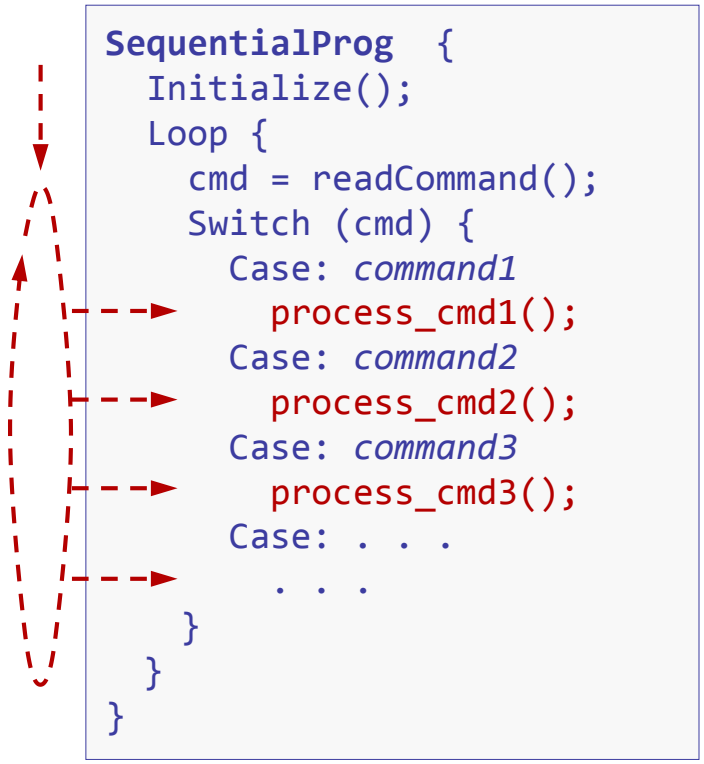
Gestion des événements

Programmation événementielle [1]

- La programmation des applications avec interfaces graphiques est généralement basée sur un paradigme nommé **programmation événementielle** *(Event Programming).*
- Dans la **programmation impérative** (séquence d'instructions), c'est le programme qui dirige les opérations (par exemple, il demande à l'utilisateur d'entrer des valeurs, calcule et affiche un résultat, etc.).
- Avec la **programmation événementielle**, ce sont les événements (généralement déclenchés par l'utilisateur, mais aussi par le système) qui pilotent l'application. Ce mode non directif convient bien à la gestion des interfaces graphiques où l'utilisateur à une grande liberté d'action (l'interface est au service de l'utilisateur et non l'inverse).
- La programmation événementielle nécessite qu'un processus (en tâche de fond) surveille constamment les actions de l'utilisateur susceptibles de déclencher des événements qui pourront être ensuite traités (ou non) par l'application (contrôleurs).

Programmation événementielle [2]

- En **programmation séquentielle**, une interface utilisateur (en lignes de commandes) pourrait être codée selon le pseudo-code suivant qui illustre le principe.



```
SequentialProg  {
  Initialize();
  Loop {
    cmd = readCommand();
    Switch (cmd) {
      Case: command1
        process_cmd1();
      Case: command2
        process_cmd2();
      Case: command3
        process_cmd3();
      Case: . . .
        . . .
    }
  }
}
```

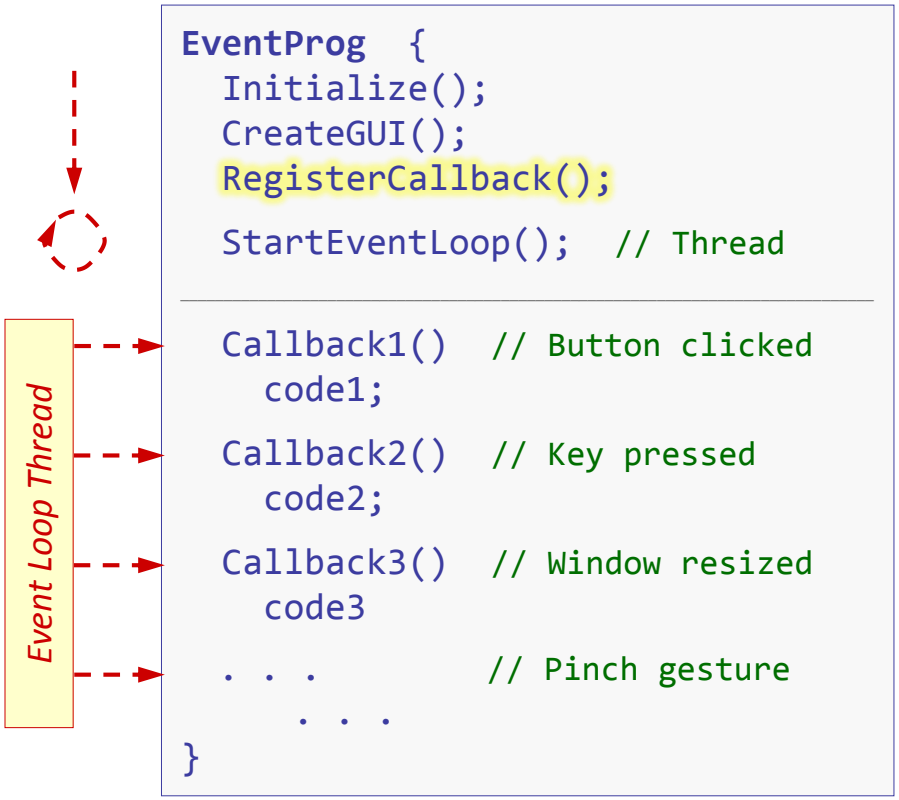
Une boucle sans fin répète le cycle suivant :

- Demande à l'utilisateur de saisir une commande *(prompt)*
- Attend que la prochaine commande soit entrée au clavier (+décodage)
- Selon la commande entrée, du code spécifique est exécuté.

Lorsque le traitement est terminé, le cycle recommence (on attend la prochaine commande).

Programmation événementielle [3]

- En **programmation événementielle**, on prépare les actions (code) à exécuter en les associant aux événements que l'on souhaite traiter (enregistrement des *callback*) et on attend que le processus de surveillance nous avertisse en exécutant le code prévu.



Après initialisation, on crée l'interface graphique puis on associe du code à chaque événement que l'on souhaite traiter *(RegisterCallback)*.

Lorsque les événements enregistrés se produisent, le code associé est automatiquement exécuté par le processus de surveillance qui tourne en tâche de fond *Event Thread*.

Événement [1]

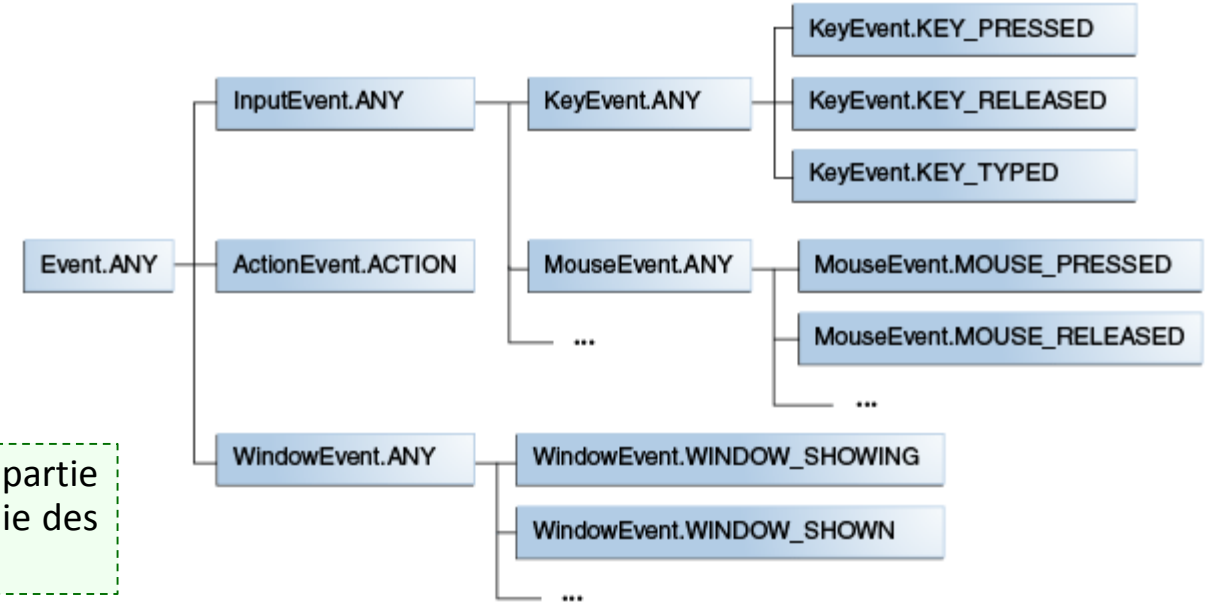
- Un **événement** *(event)* constitue une notification qui signale que quelque chose s'est passé (un fait, un acte digne d'intérêt).
- Un événement peut être provoqué par :
  - Une **action de l'utilisateur**
    - ⇨ Un clic avec la souris
    - ⇨ La pression sur une touche du clavier
    - ⇨ Le déplacement d'une fenêtre
    - ⇨ Un geste sur un écran tactile
    - ⇨ . . .
  - Un **changement provoqué par le système**
    - ⇨ Une valeur a changé (propriété)
    - ⇨ Un *timer* est arrivé à échéance
    - ⇨ Un processus a terminé un calcul
    - ⇨ Une information est arrivée par le réseau
    - ⇨ . . .

Événement [2]

- En *JavaFX* les événements sont représentés par des objets de la classe **`Event`** ou, plus généralement, d'une de ses sous-classes.
- De nombreux événements sont prédéfinis (`MouseEvent`, `KeyEvent`, `DragEvent`, `ScrollEvent`, …) mais il est également possible de créer ses propres événements en créant des sous-classes de `Event`.
- Chaque objet de type "événement" comprend (au moins) les informations suivantes :
  - Le **type de l'événement** (`EventType` consultable avec `getEventType()`)
    - ⇨ Le type permet de classifier les événements à l'intérieur d'une même classe (par exemple, la classe `KeyEvent` englobe `KEY_PRESSED`, `KEY_RELEASED`, `KEY_TYPED`)
  - La **source de l'événement** (`Object` consultable avec `getSource()`)
    - ⇨ Objet qui est à l'origine de l'événement selon la position dans la chaîne de traitement des événements *(event dispatch chain)*.
  - La **cible de l'événement** (`EventTarget` consultable avec `getTarget()`)
    - ⇨ Composant cible de l'événement (indépendamment de la position dans la chaîne de traitement des événements *(event dispatch chain)*

Types d'événements

- Chaque événement est d'un certain type (objet de type `EventType`).
- Chaque type d'événement possède un nom (`getName()`) et un type parent (`getSuperType()`).
- Les types d'événement forment donc une hiérarchie.
  - Par exemple si on presse une touche le nom de l'événement est `KEY_PRESSED` et le type parent est `KeyEvent.ANY`.
- A la racine, on a `Event.ANY` (= `EventType.ROOT`)



La figure représente une partie seulement de la hiérarchie des types d'événements.

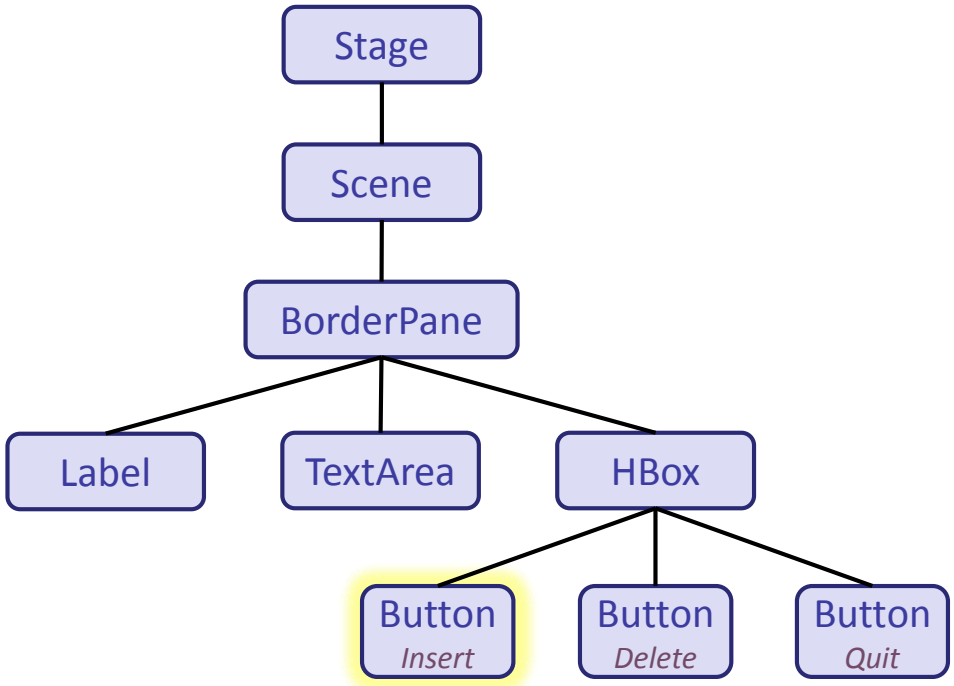
Gestion des événement [1]

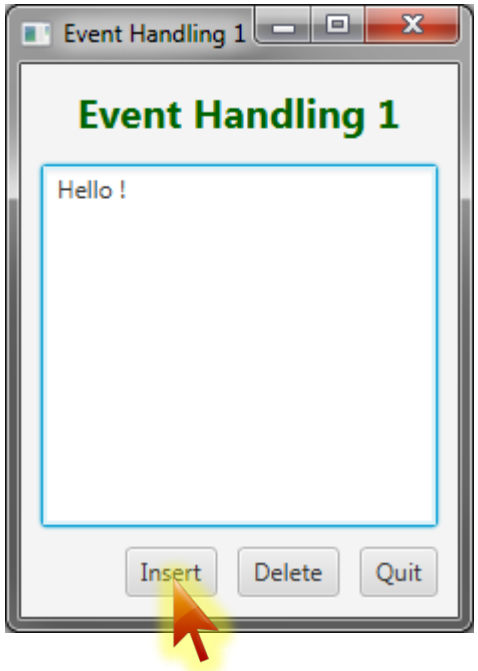
- Le traitement des événements implique les étapes suivantes :
  - La **sélection de la cible** *(Target)* de l'événement
    - ⇨ Événement clavier → le composant qui possède le focus
    - ⇨ Événement souris → le composant sur lequel se trouve le curseur
    - ⇨ Gestes continus → le composant au centre de la position initiale

      Si plusieurs composant se trouvent à un emplacement donné c'est celui qui est "au-dessus" qui est considéré comme la cible.
  - La **détermination de la chaîne de traitement** des événements (*Event Dispatch Chain* : chemin des événements dans le graphe de scène)
    - ⇨ Le chemin part de la racine *(Stage)* et va jusqu'au composant cible en parcourant tous les nœuds intermédiaires
  - Le **traitement des filtres d'événement** *(Event Filter)*
    - ⇨ Exécute le code des filtres en suivant le chemin descendant, de la racine *(Stage)* jusqu'au composant cible
  - Le **traitement des gestionnaires d'événement** *(Event Handler)*
    - ⇨ Exécute le code des gestionnaires d'événement en suivant le chemin montant, du composant cible à la racine *(Stage)*

Gestion des événement [2]

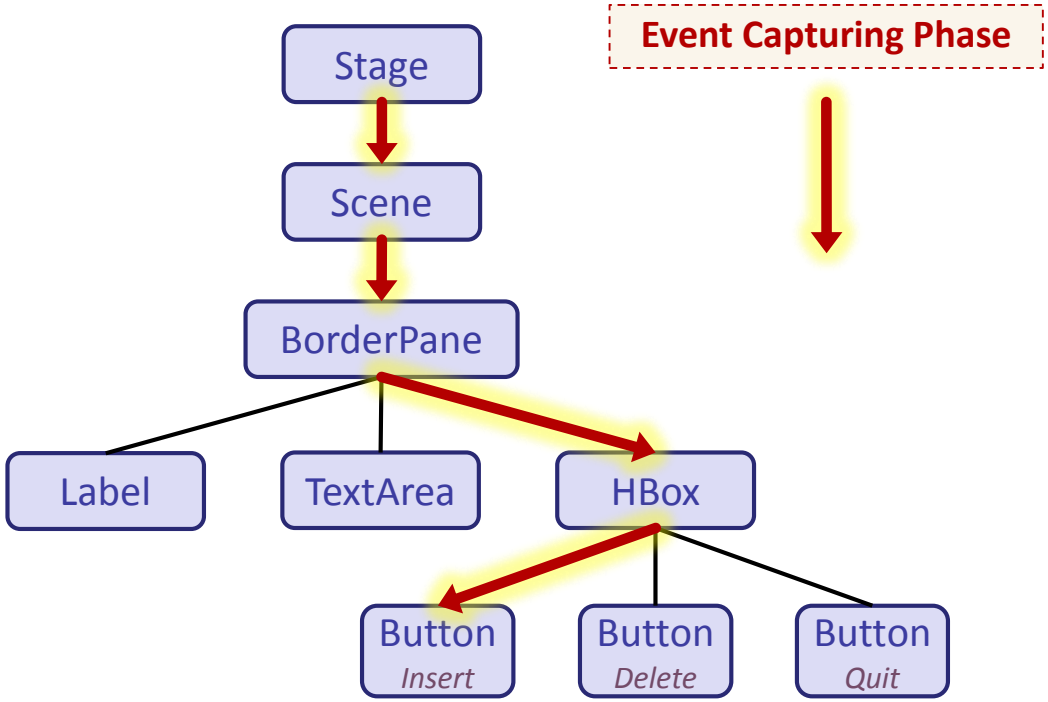
- Un exemple d'application avec son graphe de scène.
- Si l'utilisateur clique sur le bouton *Insert*, un événement de type `Action` va être déclenché et va se propager le long du chemin correspondant à la chaîne de traitement (*Event Dispatch Chain*).

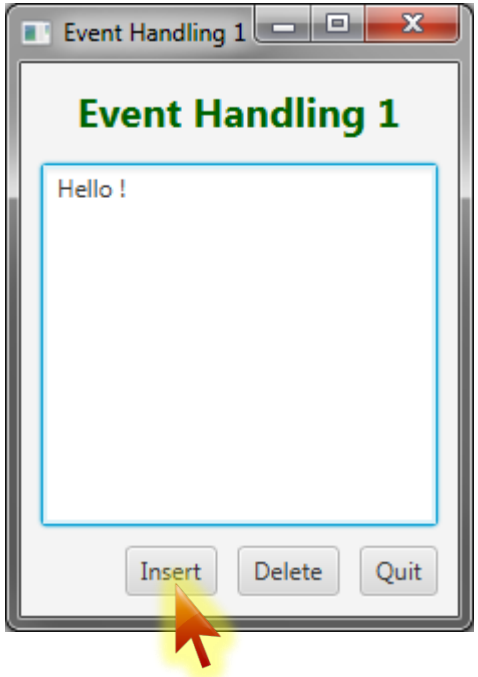




Gestion des événement [3]

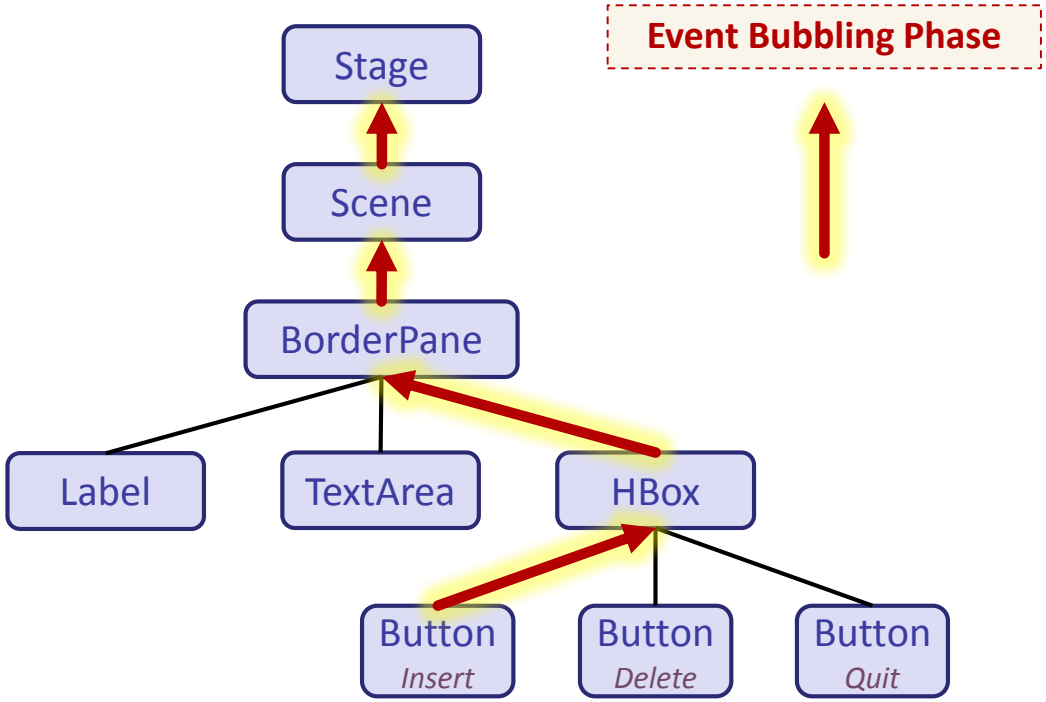
- L'événement se propage d'abord vers le bas, depuis le nœud racine *(Stage)* jusqu'à la cible *(Target)* - c'est-à-dire le bouton cliqué - et **les filtres** *(Event Filter)* éventuellement enregistrés **sont exécutés** (dans l'ordre de passage).

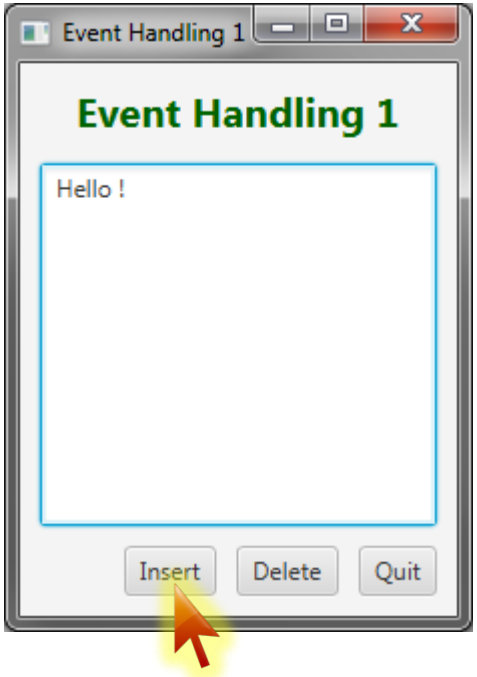




Gestion des événement [4]

- L'événement remonte ensuite depuis la cible jusqu'à la racine et **les gestionnaires d'événements** *(Event Listener)* éventuellement enregistrés **sont exécutés** (dans l'ordre de passage).





Gestion des événements [5]

- Pour gérer un événement (exécuter des instructions), il faut créer un **récepteur d'événement** ***(Event Listener)***, appelé aussi écouteur d'événement, et l'enregistrer sur les nœuds du graphe de scène où l'on souhaite intercepter l'événement et effectuer un traitement.
- Un récepteur d'événement peut être **enregistré comme filtre ou comme gestionnaire d'événement**. La différence principale entre les deux réside dans le moment où le code est exécuté :
  - Les **filtres** *(filters)* sont exécutés dans la phase descendante de la chaîne de traitement des événements (avant les gestionnaires)
  - Les **gestionnaires** *(handlers)* sont exécutés dans la phase montante de la chaîne de traitement des événements (après les filtres)
- Les filtres, comme les gestionnaires d'événements, sont des objets qui doivent implémenter l'interface fonctionnelle (générique) `EventHandler<T extends Event>` qui impose l'unique méthode `handle(T event)` qui se charge de traiter l'événement.

Gestion des événements [6]

- Pour **enregistrer un récepteur d'événement** sur un nœud du graphe de scène, on peut :
  - Utiliser la méthode **`addEventFilter()`** que possèdent tous les nœuds (les sous-classes de `Node`) et qui permet d'**enregistrer un filtre**
  - Utiliser la méthode **`addEventHandler()`** que possèdent tous les nœuds (les sous-classes de `Node`) et qui permet d'**enregistrer un gestionnaire d'événement**
  - Utiliser une des **méthodes utilitaires** *(convenience methods)* dont disposent certains composants et qui permettent d'**enregistrer un gestionnaire d'événement** en tant que propriété du composant.

    La plupart des composants disposent de méthodes nommées selon le schéma **`setOn`***`EventType`***`(`***`EventHandler`***`)`**, par exemple :

    ⇨ `setOnAction(`*`Handler`*`)`

    ⇨ `setOnKeyTyped(`*`Handler`*`)`

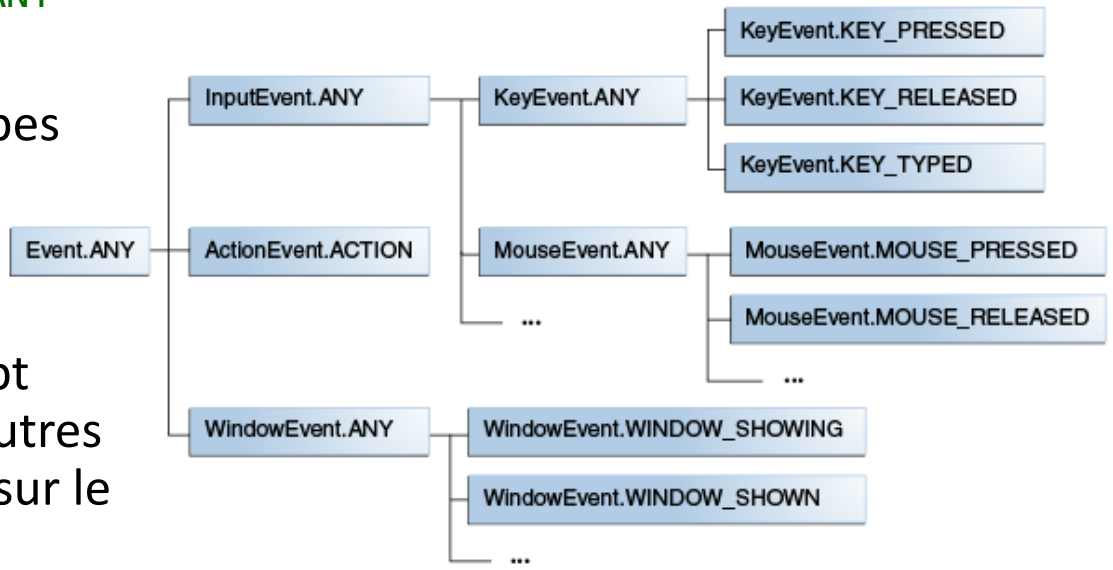
Gestion des événements [7]

- Par défaut, les événements se propagent donc le long de la chaîne de traitement *(Event Dispatch Chain)* en traversant le graphe de scène de la racine jusqu'au composant cible et retour.
- Sur chaque nœud du graphe de scène peuvent être enregistrés
  - un ou plusieurs filtres
  - un ou plusieurs gestionnaires d'événements

  qui se chargeront de traiter différents types d'événements avant de les propager au nœud suivant en parcourant la chaîne de traitement.
- Cependant, chaque récepteur d'événement (filtre ou gestionnaire) peut **interrompre la chaîne de traitement** en **consommant l'événement**, c'est-à-dire en invoquant la méthode `consume()`.
- Si un récepteur d'événement appelle la méthode `consume()`, la propagation de l'événement s'interrompt et les autres récepteurs (qui suivent dans la chaîne de traitement) ne seront plus activés.

Gestion des événements [8]

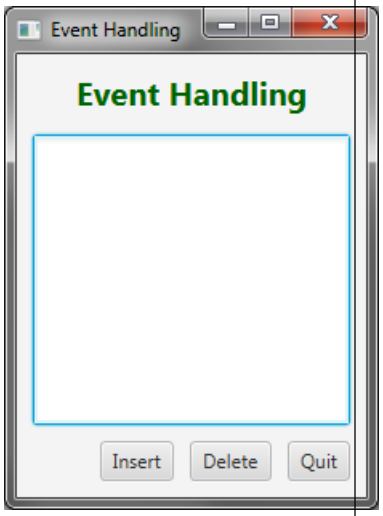
- Si un nœud du graphe de scène possède **plusieurs récepteurs** d'événements enregistrés, l'**ordre d'activation** de ces récepteurs sera basé sur la hiérarchie des types d'événement :
  - Un récepteur pour un **type spécifique** sera toujours exécuté **avant** un récepteur pour un **type plus générique**
  - Par exemple un filtre enregistré pour `MouseEvent.MOUSE_PRESSED` sera exécuté avant un filtre pour `MouseEvent.ANY` qui sera exécuté avant un filtre pour `InputEvent.ANY`
  - L'ordre d'exécution des récepteurs pour des types de même niveau n'est pas défini
  - La consommation d'un événement n'interrompt pas le traitement des autres récepteurs enregistrés sur le même nœud



Event Handling [1]

```
private BorderPane root      = new BorderPane();
private HBox       btnPanel  = new HBox(10);
private Label      lblTitle  = new Label("Event Handling");
private TextArea   txaMsg    = new TextArea();
private Button     btnInsert = new Button("Insert");
private Button     btnDelete = new Button("Delete");
private Button     btnQuit   = new Button("Quit");
```

```
primaryStage.setTitle("Event Handling");
root.setPadding(new Insets(10));

//--- Title
lblTitle.setFont(Font.font("System", FontWeight.BOLD, 20));
lblTitle.setTextFill(Color.DARKGREEN);
BorderPane.setAlignment(lblTitle, Pos.CENTER);
BorderPane.setMargin(lblTitle, new Insets(0, 0, 10, 0));
root.setTop(lblTitle);

//--- Text-Area
txaMsg.setWrapText(true);
txaMsg.setPrefColumnCount(15);
txaMsg.setPrefRowCount(10);
root.setCenter(txaMsg);
```



Event Handling [2]

```
//--- Button Panel
btnPanel.getChildren().add(btnInsert);
btnPanel.getChildren().add(btnDelete);
btnPanel.getChildren().add(btnQuit);
btnPanel.setAlignment(Pos.CENTER_RIGHT);
btnPanel.setPadding(new Insets(10, 0, 0, 0));
root.setBottom(btnPanel);

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

Code de l'interface **sans** la gestion des événements.

Event Handling [3]

- Pour traiter les événements des boutons, on peut créer une classe *contrôleur* qui implémente `EventHandler` et effectue les opérations souhaitées dans la méthode `handle()`.

```
public class InsertButtonController implements EventHandler<ActionEvent> {

  private TextArea tArea;

  //--- Constructeur ---------------------------------
  public InsertButtonController(TextArea tArea) {
    this.tArea = tArea;
  }

  //--- Code exécuté lorsque l'événement survient ----
  @Override
  public void handle(ActionEvent event) {
    tArea.appendText("A");
  }
}
```

Si on veut agir sur des composants de la vue, il faut transmettre les références nécessaires.

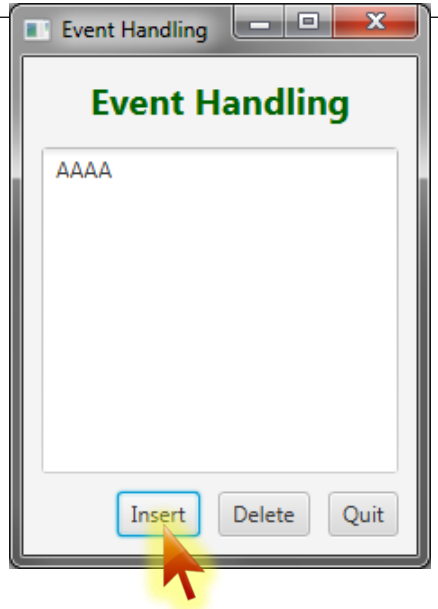
Si le code est plus complexe, on invoquera de préférence une méthode de la vue.

Event Handling [4]

- Dans la vue, il faut ensuite créer une instance de ce contrôleur et l'enregistrer comme gestionnaire d'événement (type `ACTION`) sur le bouton concerné en invoquant la méthode `addEventHandler()`.

```
. . .
//--- Button Events Handling
InsertButtonController insertCtrl = new InsertButtonController(txaMsg);

btnInsert.addEventHandler(ActionEvent.ACTION, insertCtrl);
. . .
```

- A chaque clic sur le bouton *Insert*, le gestionnaire d'événement sera exécuté et un caractère `'A'` sera ajouté dans le composant `TextArea`.

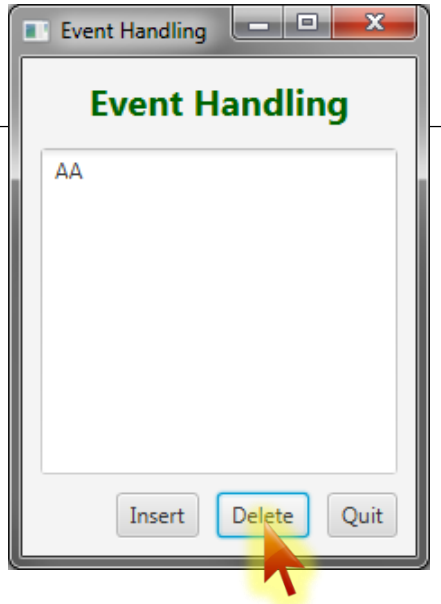


Event Handling [5]

- Une autre manière de faire consiste à créer le contrôleur sous la forme d'une classe locale anonyme. Par exemple, pour le bouton *Delete* :

```
. . .
//--- Button Events Handling
btnDelete.addEventHandler(ActionEvent.ACTION,
                          new EventHandler<ActionEvent>() {
                            @Override
                            public void handle(ActionEvent event) {
                              txaMsg.deletePreviousChar();
                            }
                          });
. . .
```

- A chaque clic sur le bouton *Delete*, le gestionnaire d'événement sera exécuté et un caractère sera supprimé dans le composant `TextArea`.



Event Handling [5]

- Une troisième possibilité pour traiter les événements des boutons, est d'utiliser la méthode `setOnAction()` et passer en paramètre une expression lambda implémentant la méthode `handle()` de l'interface `EventHandler`.
- Par exemple pour traiter les trois boutons de l'interface :

```
//--- Button Events Handling
btnInsert.setOnAction(event -> {
                                  txaMsg.appendText("A");
                                });

btnDelete.setOnAction(event -> {
                                  txaMsg.deletePreviousChar();
                                });

btnQuit.setOnAction(event ->    {
                                  Platform.exit();
                                });
```

Classe 'contrôleur'

- Dans la variante MVC synchrone, il est fréquent que la classe du contrôleur reçoive dans son constructeur les références du modèle et de la vue.

```
public class ButtonController implements EventHandler<ActionEvent> {

  private AppModel model;
  private MainView view;

  //--- Constructeur ----------------------------------------
  public ButtonController(AppModel model, MainView view) {
    this.model = model;
    this.view  = view;
  }

  //--- Code exécuté lorsque l'événement survient -------------
  @Override
  public void handle(ActionEvent event) {
    int newVal = model.getInfo();
    view.updateInfo(newVal);
  }
}
```

Références du modèle et de la vue.

Le contrôleur accède aux données du modèle et met à jour la vue.

Méthodes setOn…() [1]

- Liste des principales actions associées à des méthodes utilitaires qui permettent d'enregistrer des gestionnaires d'événements (il faut rechercher les méthodes `setOn`*EventType*`()` dans la classe).

Action de l'utilisateur	Événement	Dans classe
Pression sur une touche du clavier	`KeyEvent`	`Node`, `Scene`
Déplacement de la souris ou pression sur une de ses touches	`MouseEvent`	`Node`, `Scene`
Glisser-déposer avec la souris *(Drag-and-Drop)*	`MouseDragEvent`	`Node`, `Scene`
Glisser-déposer propre à la plateforme (geste par exemple)	`DragEvent`	`Node`, `Scene`
Composant "scrollé"	`ScrollEvent`	`Node`, `Scene`
Geste de rotation	`RotateEvent`	`Node`, `Scene`
Geste de balayage/défilement *(swipe)*	`SwipeEvent`	`Node`, `Scene`
Un composant est touché	`TouchEvent`	`Node`, `Scene`
Geste de zoom	`ZoomEvent`	`Node`, `Scene`
Activation du menu contextuel	`ContextMenuEvent`	`Node`, `Scene`

Méthodes setOn…( ) [2]

Action de l'utilisateur	Événement	Dans classe
Texte modifié (durant la saisie)	`InputMethodEvent`	`Node`, `Scene`

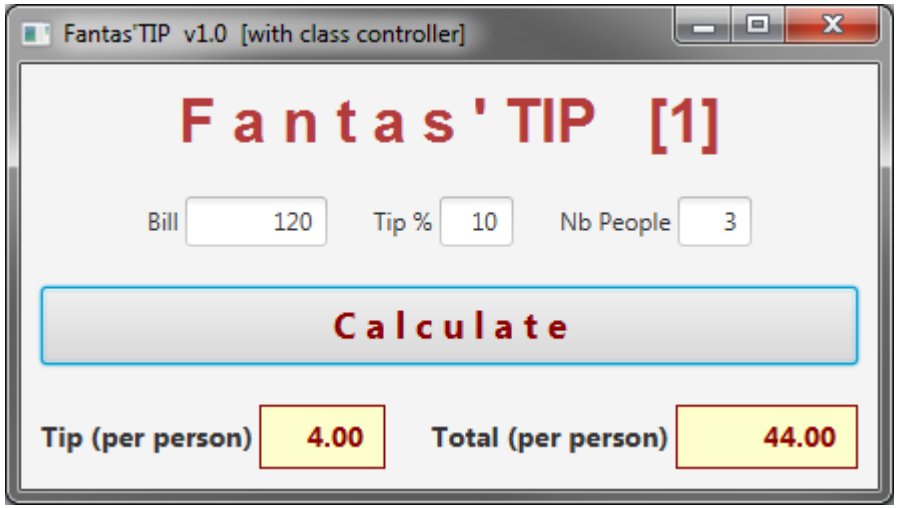
Bouton cliqué
ComboBox ouverte ou fermée
Une des options d'un menu contextuel activée
Option de menu activée
Pression sur *Enter* dans un champ texte
	`ActionEvent`	`ButtonBase`
		`ComboBoxBase`
		`ContextMenu`
		`MenuItem`
		`TextField`

Élément *(Item)* d'une liste,
… d'une table ou
… d'un arbre a été édité
	`ListView.EditEvent`
	`TableColumn.CellEditEvent`
	`TreeView.EditEvent`
		`ListView`
		`TableColumn`
		`TreeView`

Erreur survenue dans le *media-player*	`MediaErrorEvent`	`MediaView`
Menu est affiché (déroulé) ou masqué (enroulé)	`Event`	`Menu`
Fenêtre *popup* masquée	`Event`	`PopupWindow`
Onglet sélectionné ou fermé	`Event`	`Tab`
Fenêtre affichée, fermée, masquée	`WindowEvent`	`Window`

Fantas'TIP [1]

- Exemple d'application pour illustrer différentes manières de réaliser le découpage du code en exploitant plusieurs des techniques qui sont à disposition (classes `EventHandler`, expressions lambda, *binding* de propriétés, …), tout en respectant les principes de l'architecture MVC.
- L'application ***Fantas'TIP*** est un utilitaire qui calcule le pourboire à prévoir et le montant par personne, en fonction du montant de la note, du pourcentage octroyé et du nombre de convives.



Fantas'TIP [2]

- **Quatre variantes** de cette application ont été créées :
  - *Variante 1* : Avec un contrôleur réalisé sous forme de **classe 'ordinaire'**
  - *Variante 2* : Avec un contrôleur réalisé sous forme d'**expression lambda**
  - *Variante 3* : Avec un contrôleur réalisé sous forme de **liaisons de haut-niveau** *(high-level binding)* entre les données d'entrée (saisies par l'utilisateur) et les données de sortie (calculées)
  - *Variante 4* : Avec un contrôleur réalisé sous forme de **liaisons de bas-niveau** *(low-level binding)* entre les données d'entrée (saisies par l'utilisateur) et les données de sortie (calculées)
- Quelques extraits de code (les éléments importants) figurent dans les pages qui suivent.
  - L'intégralité du code source des applications est disponible sur la page :
    http://jacques.bapst.home.hefr.ch/ihm1/src/chap06_FantasTIP
  - Les applications (exécutables) sont disponibles sur la page :
    http://jacques.bapst.home.hefr.ch/ihm1/applic/fantastip

Fantas'TIP [3]

- *Variante 1* : modèle de l'application
  - Interface du modèle
  - Classe implémentant cette interface

```
public interface IFantasTipModel {
  void   setBill(double amount);
  void   setTipPercent(int percent);
  void   setNbPeople(int nbPeople);

  double getTipPerPerson();
  double getTotalPerPerson();
}
```

```
public class FantasTipModel implements IFantasTipModel {

  private double   bill         = 0;
  private int      tipPercent   = 0;
  private int      nbPeople     = 1;

  //-------------------------------------------------------------------------
  public FantasTipModel() {
  }

  //-------------------------------------------------------------------------
  @Override
  public void setBill(double amount) {
    if (amount < 0) throw new IllegalArgumentException("Amount < 0");
    this.bill = amount;
  }
```

Fantas'TIP [4]

```
  //-------------------------------------------------------------------------
  @Override
  public void setTipPercent(int percent) {
    if (percent < 0) throw new IllegalArgumentException("Percent < 0");
    this.tipPercent = percent;

  }

  //-------------------------------------------------------------------------
  @Override
  public void setNbPeople(int nbPeople) {
    if (nbPeople <= 0) throw new IllegalArgumentException("Nb people <= 0");
    this.nbPeople = nbPeople;
  }

  //-------------------------------------------------------------------------
  @Override
  public double getTipPerPerson() {
    return bill * tipPercent / 100.0 / nbPeople;
  }

  //-------------------------------------------------------------------------
  @Override
  public double getTotalPerPerson() {
    return bill/nbPeople + getTipPerPerson();
  }
}
```

Fantas'TIP [5]

- *Variante 1* : classe contrôleur (du bouton *'Calculate'*)

```
public class FantasTipController implements EventHandler<ActionEvent> {

  private IFantasTipModel model;
  private FantasTipView   view;

  //-------------------------------------------------------------------------
  // Constructor receives model and view references
  //-------------------------------------------------------------------------
  public FantasTipController(IFantasTipModel model, FantasTipView   view) {
    this.model = model;
    this.view  = view;
  }

  //-------------------------------------------------------------------------
  // Method executed when 'Calculate' button is pressed
  //-------------------------------------------------------------------------
  @Override
  public void handle(ActionEvent event) {
    //--- Get values from view (check for errors) and update model data
    try {
      double bill       = view.getBillValue();
      int    tipPercent = view.getTipPercentValue();
      int    nbPeople   = view.getNbPeopleValue();

      model.setBill(bill);
      model.setTipPercent(tipPercent);
      model.setNbPeople(nbPeople);
    }
    catch (IllegalStateException e) {     // Errors in some input values
      return;
    }

    //--- Update view with values from model output data
    view.updateTipPerPerson(model.getTipPerPerson());
    view.updateTotalPerPerson(model.getTotalPerPerson());
  }
}
```

Fantas'TIP [6]

- *Variante 1* : création d'une instance du contrôleur et association au bouton *Calculate*.
  - Ces activités sont effectuées dans le code de la vue (la méthode privée `createController()` est appelée dans la méthode `start()`).

```
. . .
//-----------------------------------------------------------------------------
// Create button controller instance and associate it to button
//-----------------------------------------------------------------------------
private void createController() {
  controller = new FantasTipController(model, this);
  btnCalc.setOnAction(controller);
}
. . .
```

Fantas'TIP [7]

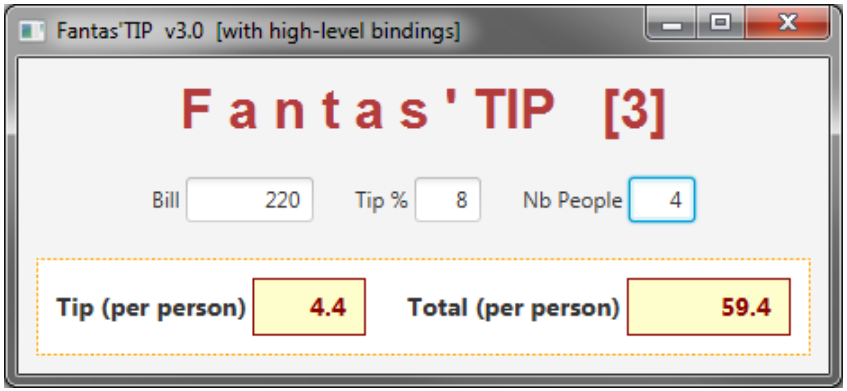
- *Variante 2* : le contrôleur est créé sous forme d'expression lambda.
  - L'expression lambda est écrite dans le code de la vue (la méthode `createController()` est appelée dans la méthode `start()`).

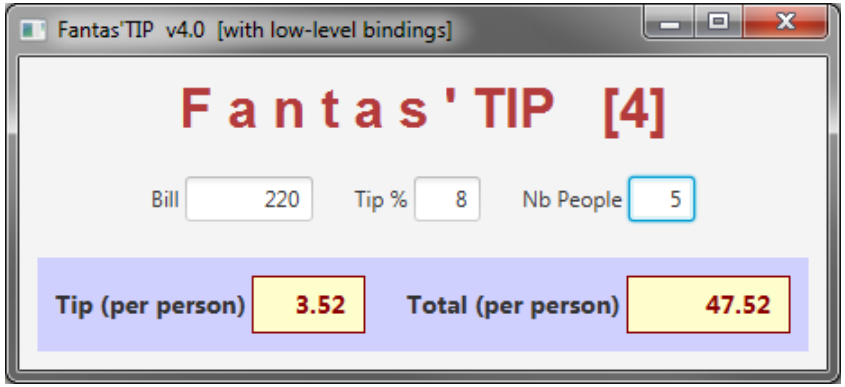
```
. . .
//-----------------------------------------------------------------------------
// Create button controller with lambda expression
//-----------------------------------------------------------------------------
private void createController() {

  btnCalc.setOnAction(event -> {
    try {
      double bill       = getBillValue();
      int    tipPercent = getTipPercentValue();
      int    nbPeople   = getNbPeopleValue();

      model.setBill(bill);
      model.setTipPercent(tipPercent);
      model.setNbPeople(nbPeople);
    }
    catch (IllegalStateException e) {   // Errors in some input values
      return;
    }

    //--- Update view with values from model output data
    updateTipPerPerson(model.getTipPerPerson());
    updateTotalPerPerson(model.getTotalPerPerson());
  });
}
```

Même si le code du contrôleur est écrit dans le fichier source de la vue, **l'expression lambda constitue bien un contrôleur** qui sera représenté par une instance d'une classe anonyme qui implémente l'interface `EventHandler`.

Fantas'TIP [8]

- Dans les *variantes 3* et *4*, les vues possèdent des liaisons (par *binding*) entre les données d'entrées et les données de sortie. L'interface n'a donc plus besoin de bouton pour déclencher le calcul.
  - Les données de sortie sont automatiquement mises à jour lorsqu'on change les données d'entrée (durant la saisie des champs texte).





Fantas'TIP [9]

- *Variante 3* : le contrôleur est constitué par les liaisons *(bindings)* créées entre les propriétés des composants d'entrée et ceux de sortie.
- Dans cette variante, on crée des liaisons dites de *haut-niveau (high-level bindings)* car les opérations sont effectuées par des invocations de méthodes en cascade. Cet enchaînement d'appels est rendu possible par l'utilisation d'un modèle de conception appelé *fluent interface pattern* ou *fluent API*.
  - Les opérations disponibles sont limitées mais suffisantes pour les calculs nécessaires dans l'application proposée.
  - Ex : `result = a.multiply(b).add(c.multiply(d));`
- Des méthodes statiques de la classe `Bindings` peuvent également être utilisées pour effectuer des opérations de haut-niveau entre des propriétés.
  - Ex :
```
result = Bindings.add(Bindings.multiply(a, b),
                      Bindings.multiply(c, d));
```

Fantas'TIP [10]

- *Variante 3* : méthode `createViewModelBindings()` appelée dans la méthode `start()` pour créer les liaisons entre les propriétés.

```
. . .
private void createViewModelBindings() {
  //--- Bind view text data with model number data (converter needed)
  tfdBill.textProperty().bindBidirectional(model.getBillPty(),
                                           dsConverter());
  tfdTipPct.textProperty().bindBidirectional(model.getTipPercentPty(),
                                             isConverter());
  tfdNbPple.textProperty().bindBidirectional(model.getNbPeoplePty(),
                                             isConverter());

  //--- Bind 'output' model properties to calculated properties
  model.getTipPerPersonPty().bind(model.getBillPty()
                                       .multiply(model.getTipPercentPty())
                                       .divide(100)
                                       .divide(model.getNbPeoplePty()));

  model.getTotalPerPersonPty().bind((model.getBillPty()
                                          .divide(model.getNbPeoplePty())
                                           .add(model.getTipPerPersonPty())));

  //--- Bind TextField properties to model properties converted to String
  tfdRTip.textProperty().bind(model.getTipPerPersonPty().asString());
  tfdRTotal.textProperty().bind((model.getTotalPerPersonPty().asString()));
}
```

Fantas'TIP [11]

- *Variante 3* : méthode `dsConverter()` utilisée pour convertir une propriété de type `double` en `String` et inversement.

```
. . .
//-----------------------------------------------------------------------------
// Number(Double) <--> String Converter
//-----------------------------------------------------------------------------
private NumberStringConverter dsConverter() {
  return new NumberStringConverter() {
    @Override
    public Number fromString(String value) {
      try {
        return Double.parseDouble(value);
      }
      catch (NumberFormatException e) {
        return Double.NaN;
      }
    }

    @Override
    public String toString() {
      return super.toString();
    }
  };
}
```

Fantas'TIP [12]

- *Variante 4* : le contrôleur est constitué par les liaisons *(bindings)* de *bas-niveau (low-level bindings)* entre les propriétés des composants d'entrée et ceux de sortie.
- Pour créer des liaisons de bas-niveau, on redéfinit les méthodes `computeValue()` de liaisons existantes (on crée des sous-classes de `IntegerBinding`, `DoubleBinding`, `StringBinding`, etc.).
  - On dispose dans ce cas de tout le potentiel des instructions et des librairies à disposition pour effectuer les calculs qui lient les propriétés (aussi complexes soient-ils).
  - Ne pas oublier de définir toutes les propriétés dont dépend la liaison en invoquant la méthode parente `super.bind(p1, p2, …)`

Fantas'TIP [13]

- *Variante 4* : méthode `dblTipPerPersonBinding()` qui retourne une spécialisation de `DoubleBinding` (qui calcule le pourboire par convive).

```
. . .
//--------------------------------------------------------------------------
// Low-level binding (calculate TipPerPerson from 'input' properties)
//--------------------------------------------------------------------------
private DoubleBinding dblTipPerPersonBinding() {

  DoubleBinding dblBinding = new DoubleBinding() {
    {
      super.bind(model.getBillPty(),
                 model.getTipPercentPty(),
                 model.getNbPeoplePty());
    }

    @Override
    protected double computeValue() {
      double tipPP = model.getBillPty().get() *
                     model.getTipPercentPty().get()/100.0 /
                     model.getNbPeoplePty().get();
      return tipPP;
    }
  };
  return dblBinding;
}
```

Fantas'TIP [14]

- *Variante 4* : méthode `strTipPerPersonBinding()` qui retourne une spécialisation de `StringBinding` (qui convertit et formate la valeur).

```
. . .
//--------------------------------------------------------------------------
// Low-level binding (convert TipPerPerson to formatted string)
//--------------------------------------------------------------------------
private StringBinding strTipPerPersonBinding() {

  StringBinding strBinding = new StringBinding() {
    {
      super.bind(model.getTipPerPersonPty());
    }

    @Override
    protected String computeValue() {
      double tipPP  = model.getTipPerPersonPty().get();
      if (tipPP < 0) return "n/a";
      String fmtRes = String.format("%.2f", tipPP);
      return fmtRes;
    }
  };
  return strBinding;
}
```

Fantas'TIP [15]

- *Variante 4* : méthode `createViewModelBindings()` qui effectue l'ensemble des liaisons entre les différentes propriétés.

```
. . .
//--------------------------------------------------------------------------
// Create all bindings (view-model-view)
//--------------------------------------------------------------------------
private void createViewModelBindings() {
  //--- Bind view text data with model number data (converter needed)
  tfdBill.textProperty().bindBidirectional(model.getBillPty(),
                                           dsConverter());
  tfdTipPct.textProperty().bindBidirectional(model.getTipPercentPty(),
                                             isConverter());
  tfdNbPple.textProperty().bindBidirectional(model.getNbPeoplePty(),
                                             isConverter());

  //--- Bind 'output' model properties to low-level calculated properties
  model.getTipPerPersonPty().bind(dblTipPerPersonBinding());
  model.getTotalPerPersonPty().bind(dblTotalPerPersonBinding());

  //--- Bind TextField properties to model properties converted to String
  tfdRTip.textProperty().bind(strTipPerPersonBinding());
  tfdRTotal.textProperty().bind(strTotalPerPersonBinding());
}
```