



Chapitre 4: **Gestion des événements**

Ce que vous allez apprendre:

A la fin de ce chapitre, vous allez apprendre à:

- Rappel sur JavaFX
- Structurer une application suivant le schéma Modèle Vue Contrôleur (MVC)
- Développer des interfaces graphiques avec le langage FXML
- Utiliser le SceneBuilder pour dessiner l'interface utilisateur
- Gestion d'événements avec Scene Builder





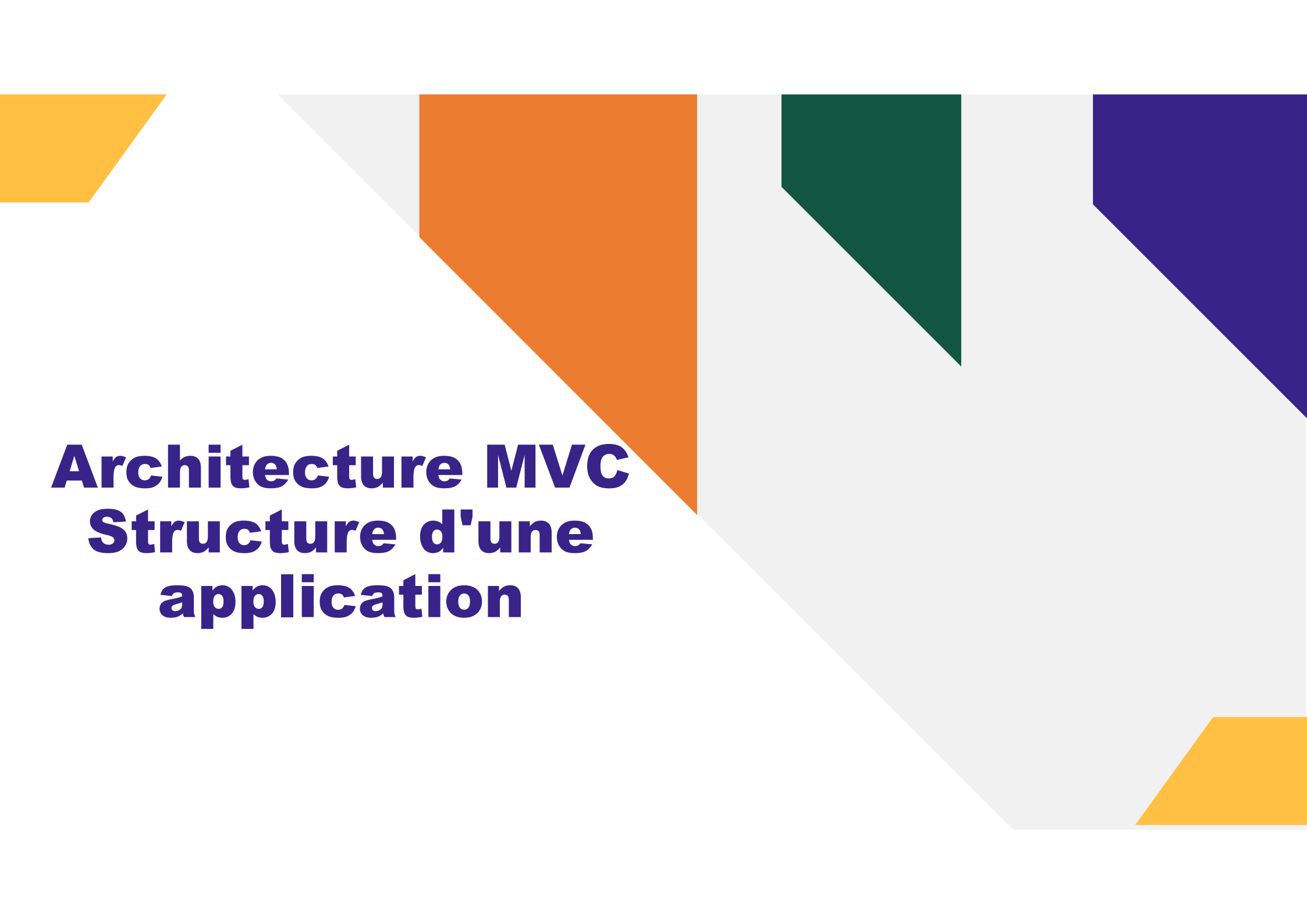
Rappel sur JavaFX

Rappel sur JavaFX

- Les 3 classes de base d'une application JavaFX : **Application, Stage et Scene**
- Pour démarrer une application, on procède tjrs de la même manière:
 - La classe principale d'une application javaFX hérite de la classe **javafx.application.Application**. C'est la méthode `start()` le point d'entrée pour toute application JavaFX.
 - Cette classe **javafx.application.Application** gère tout le cycle de vie de l'application pour vous (ouverture des fenêtres, initialisations, le démarrage et la fin de l'application, etc).
 - Après avoir exécuté la méthode **Application.launch()** –(dans le `main()`) - l'application JavaFX s'initialise et appelle la méthode `start()` pour démarrer.

Programmation procédurale vs déclarative

- JavaFX offre deux techniques complémentaires pour créer les interfaces (I/F) graphiques des applications :
 - **Manière procédurale**
 - Utilisation d'API pour construire l'interface avec du code Java
 - Création et manipulation dynamique des interfaces
 - Création d'extensions et variantes (par héritage)
 - **Manière déclarative**
 - En décrivant l'interface dans un fichier FXML (syntaxe XML)
 - L'utilitaire graphique Scene Builder facilite la création et la gestion des fichiers FXML
 - L'interface peut être créée par un designer (sans connaissance Java, ou presque...)
 - Séparation entre présentation et logique de l'application (MVC)
- Il est possible de mélanger les deux techniques au sein d'une même application (l'API [javafx.fxml](#) permet de faire le lien entre les deux).




Architecture MVC

Structure d'une application

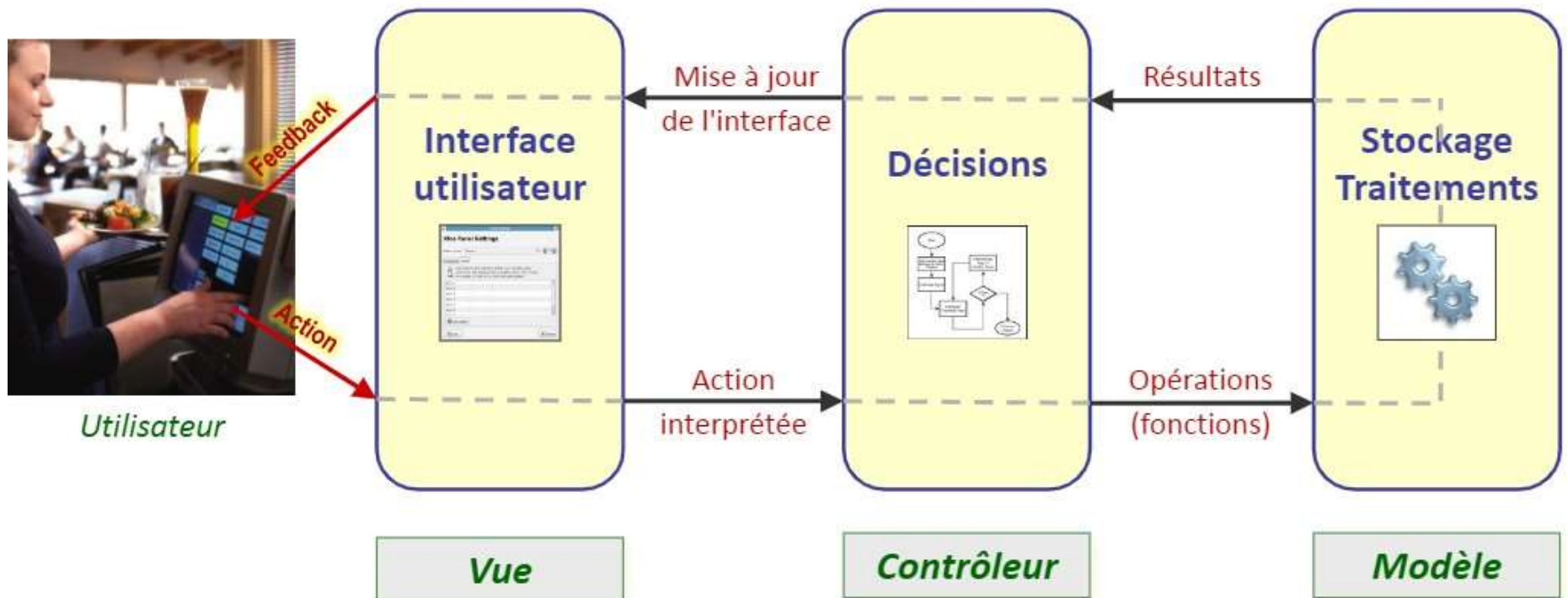


Architecture MVC

- 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 **Model -View-Controller**.
 - 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.).
- 

Interactions MVC

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



Structure d'une application javaFx


- 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.

Fichiers FXML (1)

- Au centre de l'approche déclarative, se trouve les **fichiers FXML**.
- Un fichier FXML est un fichier au format XML dont la syntaxe est conçue pour décrire l'interface (**la vue**) avec ses composants, ses conteneurs, sa disposition, ...
- A l'exécution, le fichier FXML sera chargé par l'application (classe **FXMLLoader**) et un objet Java sera créé (généralement la racine est un conteneur) avec les éléments que le fichier décrit (les composants, conteneurs, graphiques, ...).
 - Un fichier FXML constitue une forme particulière de sérialisation d'objets, utilisée spécifiquement pour décrire les interfaces
- Il est possible de créer les fichiers FXML avec un éditeur de texte mais, plus généralement, on utilise un outil graphique (**SceneBuilder**) qui permet de concevoir l'interface de manière conviviale et de générer automatiquement le fichier FXML correspondant.




Fichiers FXML (2)

- Les objets créés par le chargement de fichiers FXML peuvent être assignés à la **racine d'un graphe de scène** ou représenter **un des nœuds** dans un graphe de scène créé de manière procédurale.
 - Une fois chargés, les nœuds issus de fichiers FXML sont totalement équivalents à ceux créés de manière procédurale. Les mêmes opérations et manipulations peuvent leur être appliquées.
 - Le langage FXML n'est pas associé à un schéma XML mais la structure de sa syntaxe correspond à celle des API JavaFX:
 - Les **classes** JavaFX(conteneurs, composants) peuvent être utilisées comme **éléments** dans la syntaxe XML
 - Les **propriétés** des composants correspondent à leurs **attributs**
- 

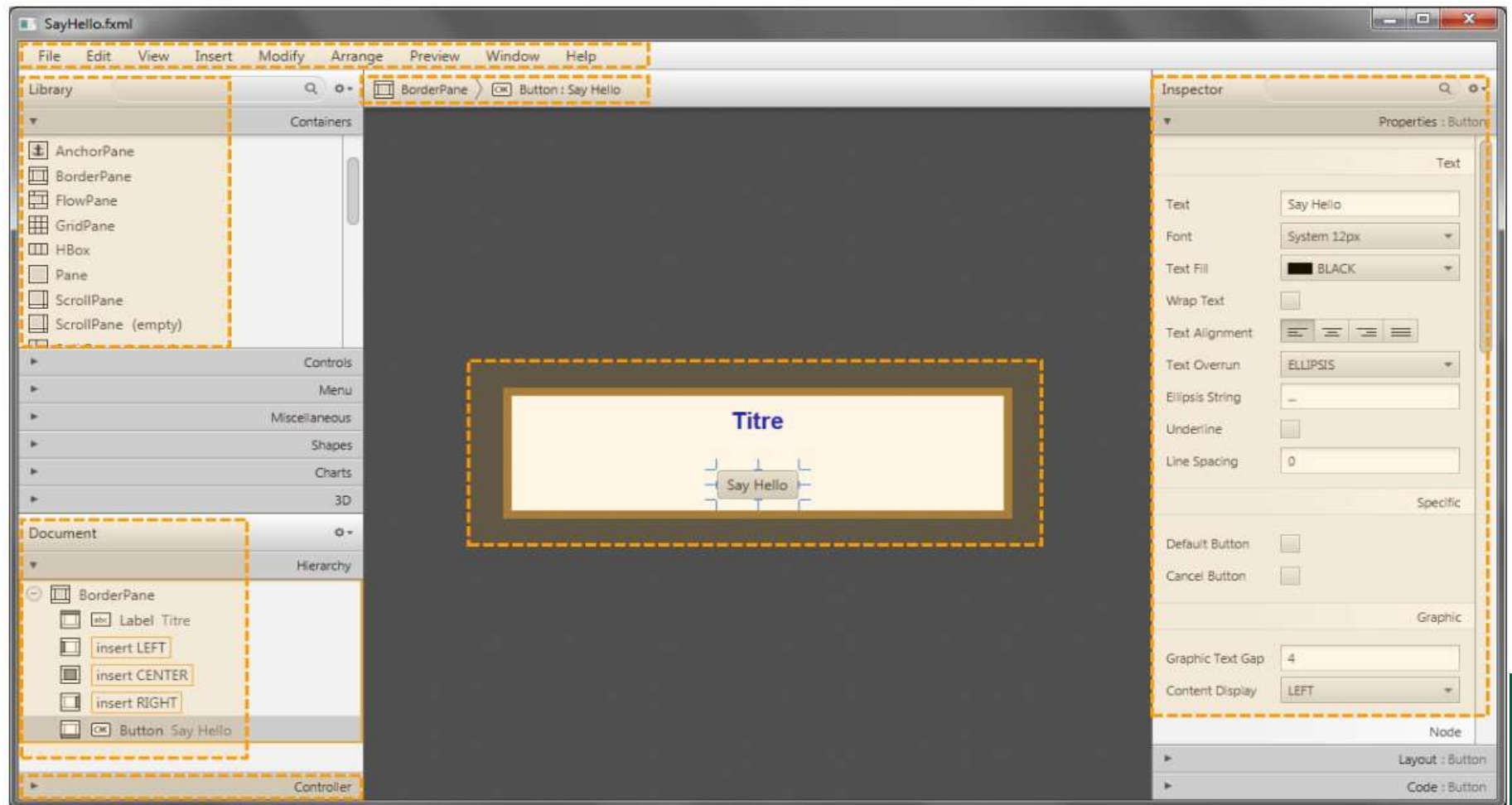


SceneBuilder (1)

- L'outil graphique SceneBuilder qui permet, de manière interactive(WYSIWYG), de concevoir les interfaces (les vues) en utilisant notamment de créer les fichiers FXML. Ceci en assemblant les conteneurs et les composants et en définissant leurs propriétés.
 - SceneBuilder est une application qui doit être installée disponible sur le lien suivant: <https://gluonhq.com/products/scene-builder/#download>
 - Le mode de fonctionnement de cet utilitaire est assez classique avec une zone d'édition centrale, entourée d'un certain nombre d'outils : palettes de conteneurs, de composants, de menus, de graphiques, vue de la structure hiérarchique de l'interface, inspecteurs de propriétés, de layout, etc.
- 

SceneBuilder (2)

Aperçu de l'écran principal :






Les événements : Définition et méthodes de gestion

Gestion des événements en JavaFX

- En JavaFX, le principe général qui est utilisé est la programmation dite **événementielle**.
- La **programmation événementielle** est un paradigme de programmation conçu pour la programmation avec interfaces graphiques et qui est fondé sur les **événements**.
- Le principe de la programmation événementielle est qu'en permanence une tâche (un petit programme) surveille l'exécution du programme et agit lorsqu'un événement survient.
- Autrement dit avec JavaFX le programme attend les actions (**input events**) de l'utilisateur
- 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 a une grande liberté d'action (l'interface est au service de l'utilisateur et non l'inverse).



Définition d'un événement (1)

- **Un événement (event)** constitue une notification qui signale que quelque chose s'est passé (un fait, un acte intéressante).
 - 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, etc.
 - **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, etc.
- 

Définition d'un évènement (2)

- Un évènement (**Event**) dans une application à interface graphique est l'occurrence d'une interaction entre l'utilisateur et l'application;

Exemple : un clic avec la souris, la pression avec une touche de clavier, le déplacement d'une fenêtre, un geste sur une écran tactile, etc.

- Un évènement dans JavaFx est représenté par un objet de la classe `Javafx.event.Event` ou de l'une de ses sous-classes;

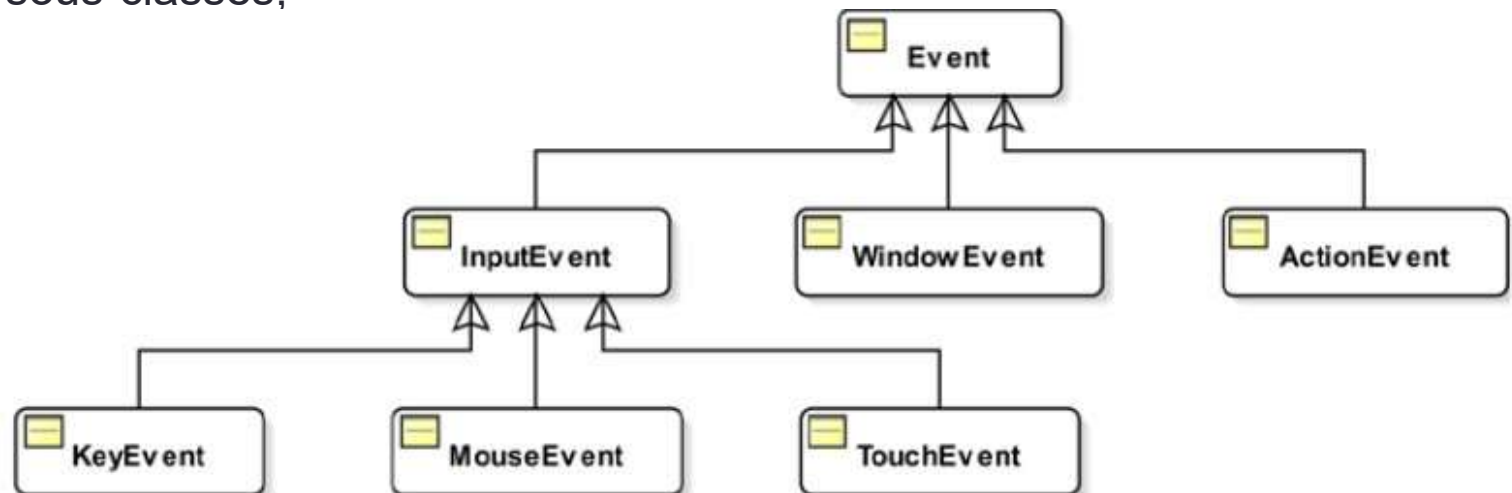
WindowEvent: Un évènement de la fenêtre : l'affichage ou le masquage d'une Fenêtre

ActionEvent: plusieurs types d'événements représentant différents type d'actions

KeyEvent: Une entrée utilisateur à partir du clavier

MouseEvent: Une entrée utilisateur à partir de la souris

TouchEvent: Une entrée utilisateur en touchant l'écran



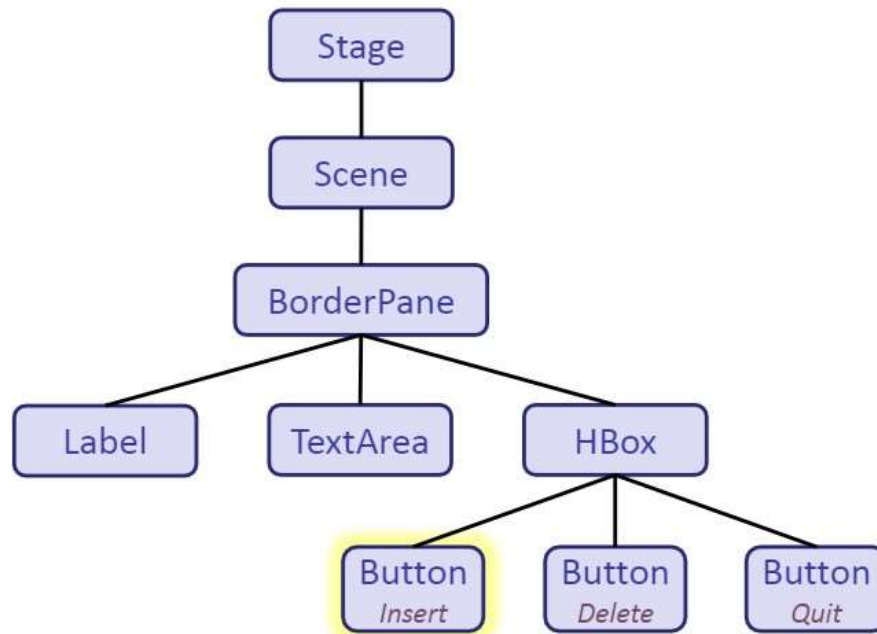
Un diagramme de classe partiel de la classe **Javafx.event.Event**

Définition d'un événement (3)

- 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.
 - ✓ **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)

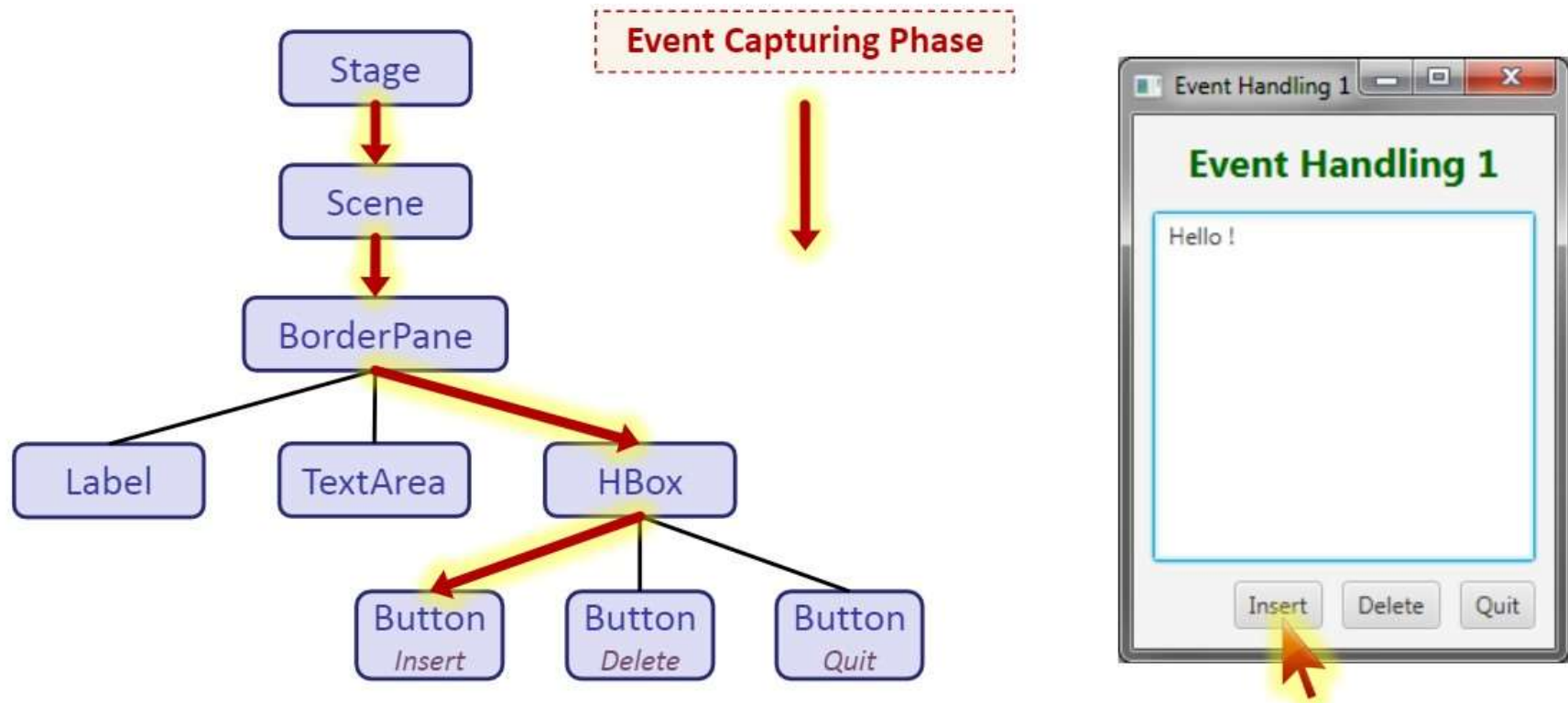
Exemple d'application (1)

- 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).



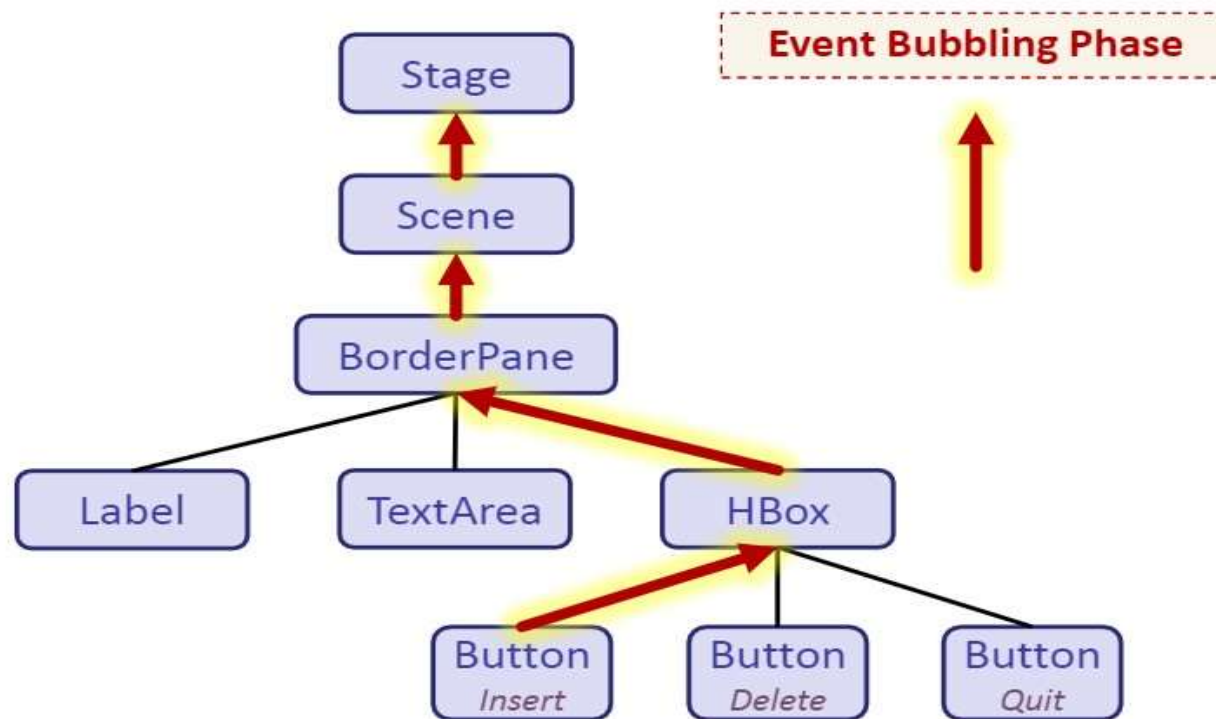
Exemple d'application (2)

- 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 (EventFilter) éventuellement enregistrés sont exécutés(dans l'ordre de passage).



Exemple d'application (3)

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






Gestion des événements [1]

- **Définition d'un événement:**

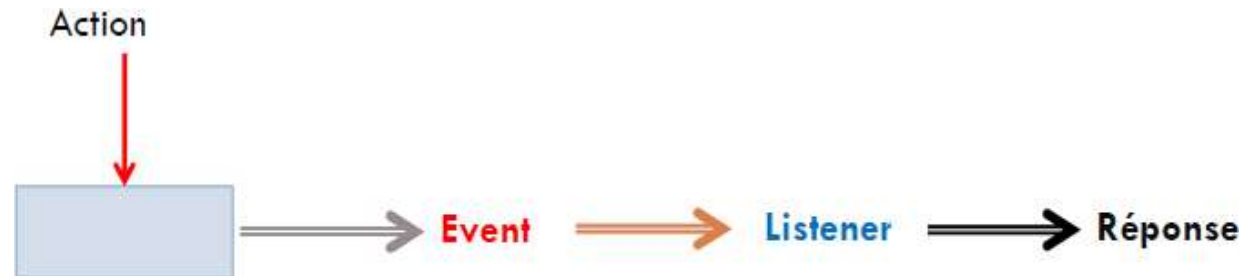
- Message à destination de l'application :
 - Existence d'une action
- Informations spécifiques à l'action

Provenant soit

- d'une action utilisateur (saisie clavier, click souris, ...).
 - de l'application elle-même (exécution d'un Timer).
- 

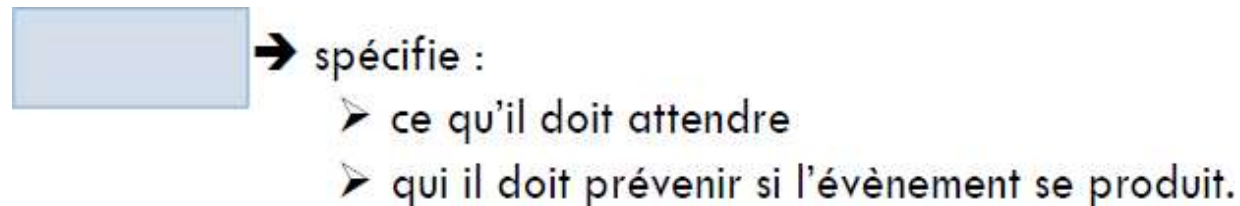
Gestion des événements [2]

- Mécanisme:




- Trois types d'objets:

- ✓ L'objet qui reçoit l'événement (Button par exemple)
- ✓ L'événement en lui même (Event)
- ✓ L'objet qui traite cet événement (Listener ou écouteur); (notre Application, par ex.)





Gestion des événements [3]

- Pour gérer un événement (exécuter des instructions), il faut créer un **récepteur d'événement (EventListener)**, 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.
 - Pour enregistrer un récepteur d'événement sur un nœud de graphe de scène, on peut utiliser **3 méthodes**.
- 

Gestion des événements [4]

- Pour enregistrer un récepteur d'événement (listener) sur un nœud du graphe de scène, on peut:
 - **Méthode 1**: Créer une classe qui implémente la classe `EventHandler` et lui faire appel dans la méthode `start`
 - **Méthode 2**: Créer une classe anonyme et y ajouter la méthode `addEventHandler()` qui permet d'enregistrer un gestionnaire d'événement (handler)
 - **Méthode 3**: Utiliser une des méthodes utilitaires dont disposent certains composants et qui permettent d'enregistrer un gestionnaire d'événement en tant que propriété du composant. Ces méthodes vont être utilisés dans le corps de la méthode « `initialize..` » de la classe `controller`.
- La plupart des composants disposent de méthodes nommées selon le schéma `setOnEventType(EventHandler)`, par exemple :
 - ✓ `setOnAction(handler)`
 - ✓ `setOnKeyTyped(handler)`

Méthode 1 de gestion d'événements

- Tout D'abord créer l'interface en question avec sceneBuilder
- Pour traiter les événements des boutons, on peut créer une classe qui implémente **EventHandler** et effectue le traitement souhaité 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



Méthode 1 de gestion d'événements

- Dans la méthode start , 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().

```
class Main extends Application {  
    public void start( Stage primaryStage) {  
        TextArea txtAMsg = new TextArea();  
  
        * * *  
        //--- Button Events Handling  
        InsertButtonController insertCtrl = new InsertButtonController(txtAMsg);  
        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.



Méthode 2 de gestion d'événements

- Créer une classe locale anonyme dans la méthode start : c'est une classe locale qui ne porte pas de nom
- Elle permet de déclarer une classe et de créer un objet de celle-ci en une expression.
- La classe anonyme est un sous type d'une interface ou d'une classe abstraite ou Concrète.

```
Type var = new Type(param1, param2,...) {  
    // définition des membres  
    // méthodes  
}
```

Exemple

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



Méthode 3 de gestion d'événements

- Utiliser les méthodes utilitaires (setOn..), utilisées aussi dans l'approche déclarative (avec utilisation de sceneBuilder)
- Utiliser une expression lambda, qui peut être assimilée à une fonction anonyme qui a potentiellement accès au contexte du code englobant
 - Il s'agit essentiellement d'un bloc de code avec des paramètres et qui est destiné à être ultérieurement exécuté.
 - La syntaxe de base est:

(parametres) → {statements;}

□ **Exemple:**

```
btnInsert.setOnAction(event -> {  
    txAMsg.appendText("A");  
});
```



Méthode 3 de gestion d'événements

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 (<i>Drag-and-Drop</i>)	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 (<i>swipe</i>)	SwipeEvent	Node, Scene
Un composant est touché	TouchEvent	Node, Scene
Geste de zoom	ZoomEvent	Node, Scene
Activation du menu contextuel	ContextMenuEvent	Node, Scene

Liste des principales actions associées à des méthodes utilitaires (**setOn...**) qui permettent d'enregistrer des gestionnaires d'événements.

Action de l'utilisateur	Événement	Dans classe
Texte modifié (durant la saisie)	InputMethodEvent	Node, Scene
Bouton cliqué	ActionEvent	ButtonBase
ComboBox ouverte ou fermée		ComboBoxBase
Une des options d'un menu contextuel activée		ContextMenu
Option de menu activée		MenuItem
Pression sur <i>Enter</i> dans un champ texte		TextField
Élément (<i>Item</i>) d'une liste,	ListView. EditEvent	ListView
... d'une table ou	TableColumn. CellEditEvent	TableColumn
... d'un arbre a été édité	TreeView. EditEvent	TreeView
Erreur survenue dans le <i>media-player</i>	MediaErrorEvent	MediaView
Menu est affiché (déroulé) ou masqué (enroulé)	Event	Menu
Fenêtre <i>popup</i> masquée	Event	PopupWindow
Onglet sélectionné ou fermé	Event	Tab
Fenêtre affichée, fermée, masquée	WindowEvent	Window

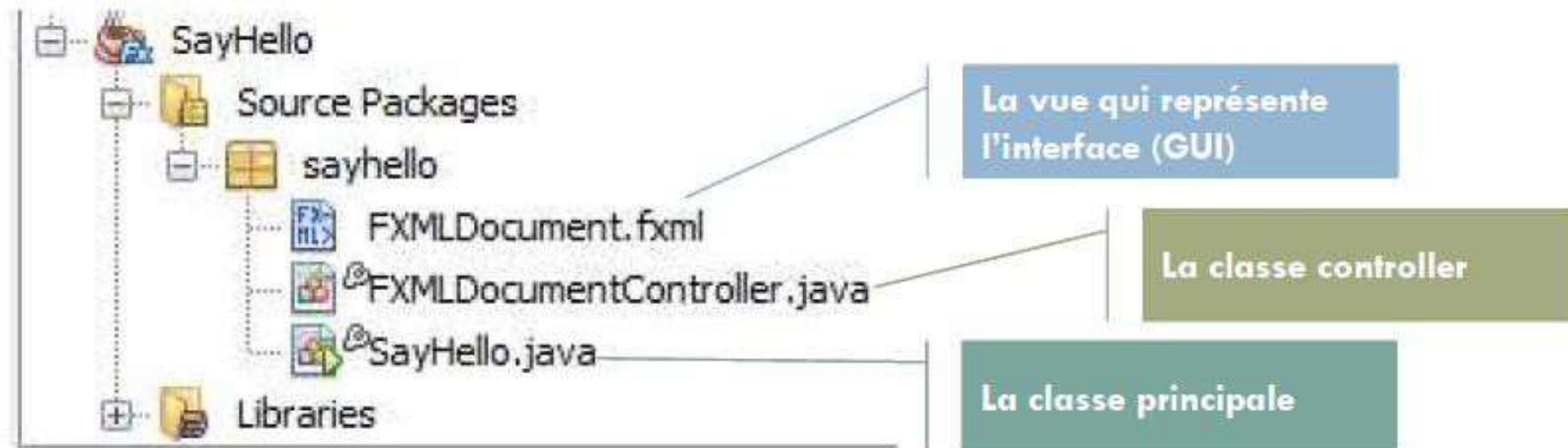


Gestion d'événements avec Scene Builder

- 1. Création d'un projet FXML**
- 2. Liens FXML ↔ Programme**

Étape 1: Création d'un projet FXML

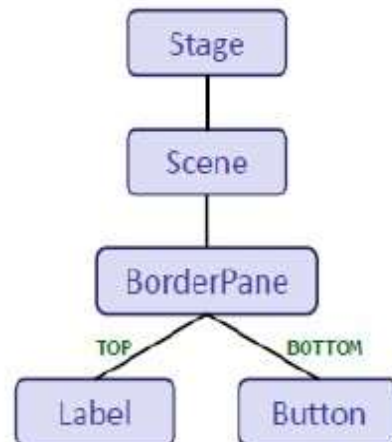
- Soit le projet FXML suivant:



Étape 1: Création d'un projet FXML: Projet « SayHello »

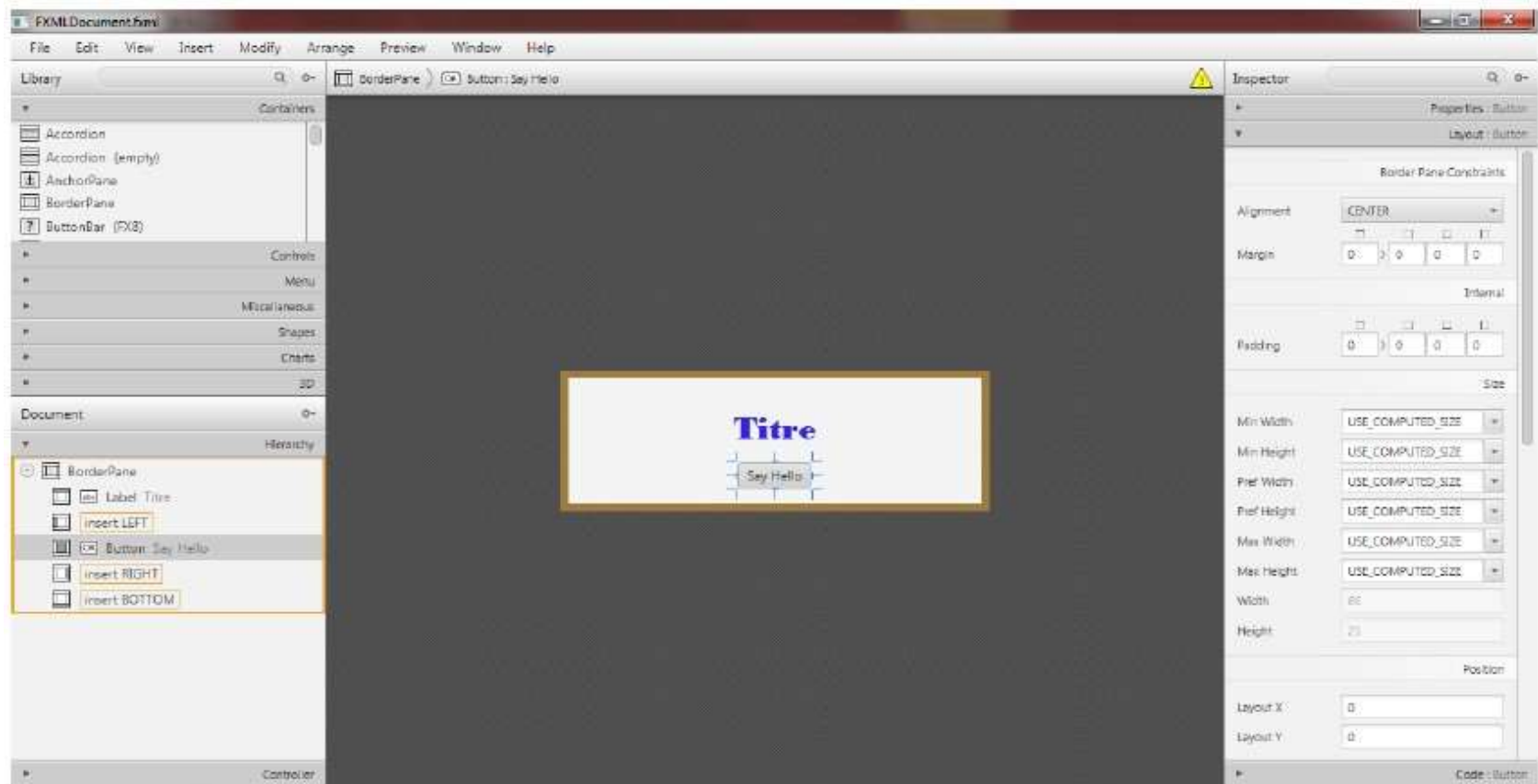
- Un exemple d'application très simple :
 - Un conteneur **BorderPane**
 - Deux composants : **Label** et **Button**
 - Quelques adaptations de propriétés (taille, couleur, marge, ...)

Graphe de scène et apparence finale de l'application



La vue: FXMLDocument.fxml [1]

- Si on clique bouton droit et open sur le fichier FXMLDocument.fxml, le scene Builder s'ouvre, et en voici l'interface à créer



La vue: ihm1.fxml [2]

- Le code source du fichier ihm1.fxml:

```
<BorderPane prefHeight="80.0" prefWidth="250.0"
    style="-fx-background-color: #FFFCAA;"
    xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="supp_cours.chap07.SayHelloController">
    <top>
        <Label id="title" fx:id="title" text="Titre" textFill="#3723e8 "
            BorderPane.alignment="CENTER">
            <font>
                <Font name="SansSerif Bold" size="20.0" />
            </font>
        </Label>
    </top>
    <bottom>
        <Button fx:id="btnHello" onAction="#handleButtonAction" text="Say Hello"
            BorderPane.alignment="CENTER" />
    </bottom>
    <padding>
        <Insets bottom="10.0" left="5.0" right="5.0" top="10.0" />
    </padding>
</BorderPane>
```



La classe principale: SayHello.java

- La méthode start() de la classe principale peut charger le fichier
- La méthode getResource(name) de la classe Class permet de trouver (par le classloader) l'URL d'une ressource à partir de son nom.

```
public void start(Stage stage) throws Exception {  
  
    // Chargement du fichier fxml  
    FXMLLoader loader = new FXMLLoader (getClass().getResource("FXMLDocument.fxml"));  
    Parent root = loader.load();  
  
    Scene scene = new Scene(root);  
  
    stage.setTitle("SayHello FXML");  
    stage.setScene(scene);  
    stage.show();  
}
```

La classe controller: SayHelloController.java

- Dans la variante déclarative, une classe séparée joue le rôle de contrôleur pour traiter l'action du clic sur le bouton.

```
public class SayHelloController {  
    @FXML  
    private Button btnHello; // Object injected by FXMLLoader (fx:id="btnHello")  
  
    @FXML  
    private Label title; // Object injected by FXMLLoader (fx:id="title")  
  
    @FXML  
    private void handleButtonAction(ActionEvent event) {  
        title.setText("H e l l o  !");  
        title.setTextFill(Color.FUCHSIA);  
    }  
}
```

Résultat attendu
lors du clic sur le
bouton « Say
Hello »



Étape 2: Liens FXML ↔ Programme [1]

- Pour la gestion d'évènements, on est amené à créer des liens entre le fichier FXML et le programme java

1ère étape:

- Affecter à chaque composant (**cible et source de l'événement**) un identifiant « **fx-id** » dans le scene builder
- Ajouter une action sur l'**objet source** de l'événement (sur le bouton par exemple) dans le scene builder
- Affecter au **controller**, la classe correspondante dans le projet en cours dans le scene builder

2ème étape:

- Déclarer des attributs correspondant aux différents composants ayant un fx-id précédés par l'annotation **@FXML**
 - Ajouter une méthode qui traite l'évènement dans la classe controller
- Toutes ces étapes seront détaillées dans les diapositives qui suivent:

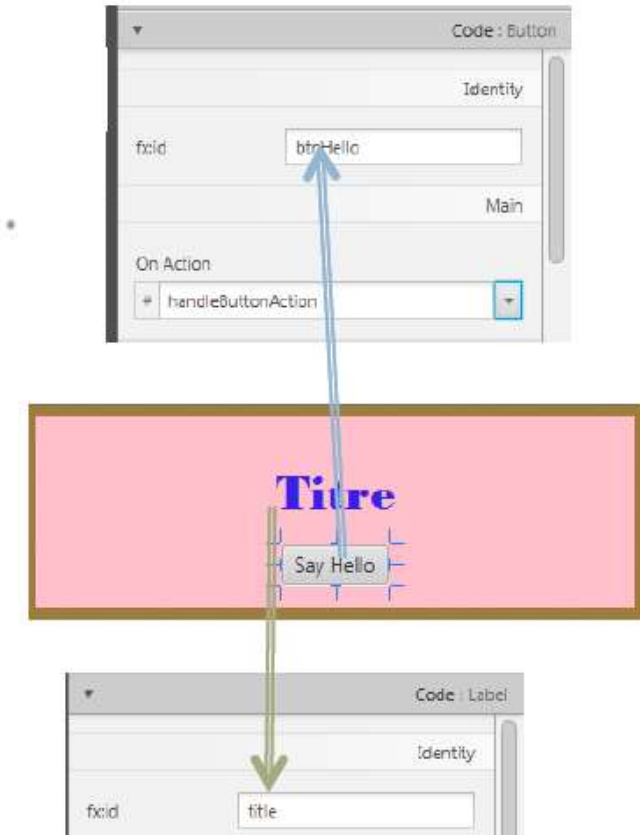
Étape 2: Liens FXML ↔ Programme [2]

- Le lien entre les composants décrits dans le fichier FXML et le programme est établi par les **attributs fx:id**:

```
<Label id="title" fx:id="title" text="Titre" textFill="#0022cc" ..
```

- L'attribut fx:id fonctionne en lien avec l'annotation **@FXML** que l'on peut utiliser dans les contrôleurs, et qui va indiquer au système que le composant avec le nom **fx:id** pourra être *injecté* dans l'objet correspondant de la classe contrôleur

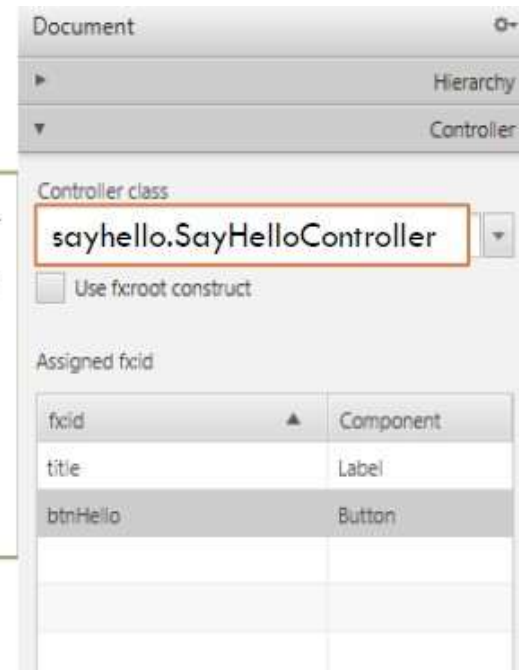
```
public class SayHelloController implements Initializable {  
  
    @FXML  
    private Button btnHello;  
  
    @FXML  
    private Label title;  
}
```



Étape 2: Liens FXML ↔ Programme [2]

- La classe qui joue le rôle de contrôleur pour une interface déclarée en FXML doit être annoncée dans l'élément racine, en utilisant l'attribut **fx:controller**:

```
<BorderPane prefHeight="118.0" prefWidth="370.0" style="-fx-background-color: pink;"  
xmlns="http://javafx.com/javafx/8.0.65"  
xmlns:fx="http://javafx.com/fxml/1"  
fx:controller="sayhello.SayHelloController">
```



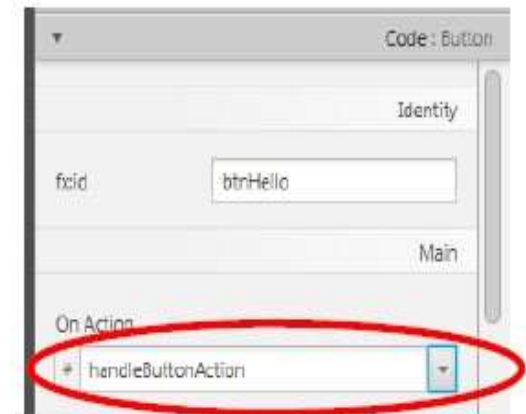
Étape 2: Liens FXML ↔ Programme [3]

- Pour les composants actifs déclarés dans une interface en FXML, on peut indiquer la méthode du contrôleur qui doit être invoquée en utilisant l'attribut `fx:onEvent="#methodName"`:

```
<Button fx:id="btnHello" onAction="#handleButtonAction"  
        text="Say Hello" BorderPane.alignment="CENTER" />
```

- Dans la classe contrôleur, ces méthodes devront (comme les composants associés) être annotées avec `@FXML`.

```
@FXML  
private void handleButtonAction(ActionEvent event) {  
    title.setText("H e l l o  !");  
    title.setTextFill(Color.FUCHSIA);  
}
```



Étape 2: Liens FXML ↔ Programme [4]

- Dans les classes qui agissent comme "contrôleurs", on peut définir une méthode `initialize()` (qui doit être annotée avec `@FXML`) pour effectuer certaines initialisations.
- Cette méthode est automatiquement invoquée après le chargement du fichier FXML.
- Elle peut être utile pour initialiser certains composants, en faisant par exemple appel au modèle.

```
    * * *  
    @FXML  
    private void initialize() {  
        cbbCountry.getItems().addAll("Allemagne", "Angleterre", "Belgique",  
                                     "Espagne",    "France",    "Italie",  
                                     "Pays-Bas",   "Portugal",   "Suisse");  
  
        lstProducts.getItems().addAll(model.getProducts());  
        * * *  
    }
```

Étape 2: Liens FXML ↔ Programme [5]

- Pour gérer un événement, on déjà mentionné qu'on peut utiliser les méthodes utilitaires `setOnEventType` dans la méthode `initialize(...)` de la classe `controller`

1ère étape:

- Affecter aux composants cible et source de l'événement un « fx-id » dans le scene builder
- Affecter au controller, la classe correspondante dans le projet en cours dans le scene builder

2ème étape:

- Déclarer des attributs correspondant aux différents composants ayant un fx-id précédés par l'annotation `@FXML`
- Appliquer la méthode `setOnEventType` sur le composant actif dans le corps de la méthode `initialize` du controller.

Étape 2: Liens FXML ↔ Programme [6]

- Exemple appliqué dans le projet *SayHello* créée précédemment:

```
public class FXMLDocumentController implements Initializable {  
  
    @FXML  
    private Button btnHello;  
  
    @FXML  
    private Label title;  
  
    @Override  
    public void initialize(URL url, ResourceBundle rb) {  
        btnHello.setOnAction(new EventHandler<ActionEvent>() {  
            @Override  
            public void handle(ActionEvent e) {  
                title.setText("H e l l o !");  
                title.setTextFill(Color.FUCHSIA);  
            }  
        });  
    }  
}
```

Equivalent: utilisation
d'expression **lambda**

```
public void initialize(URL ur,  
    ResourceBundle rb) {  
    Btn.setOnAction (event -> {  
        Title.setText (« Hello! »);  
        Title.setTextFill (Color.FUSHIA);  
    });  
}
```