**TP2(partie 1)**

Activation des composants d’une interface - partie 1

En général, la construction d’une interface graphique (ou GUI) passe par 2 étapes :

1. Organisation et affichage des composants de l’interface (voir TP1)
2. **Activation des composants de l’interface**

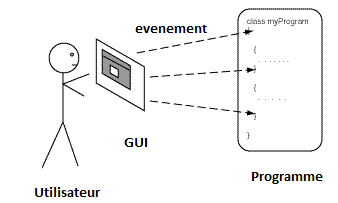
L’objectif de ce TP est de maitriser la 2eme partie : **Activation des composants.**

1. **Evènements et programmation évènementielle**

Une GUI affiche à l’utilisateur plusieurs composants. L’utilisateur contrôle l’application en interagissant avec les composants à travers des actions comme :

* Cliquer sur un bouton pour choisir une option
* Faire un choix sur un menu
* Saisir du texte sur un champ de texte
* Déplacer une barre de défilement…

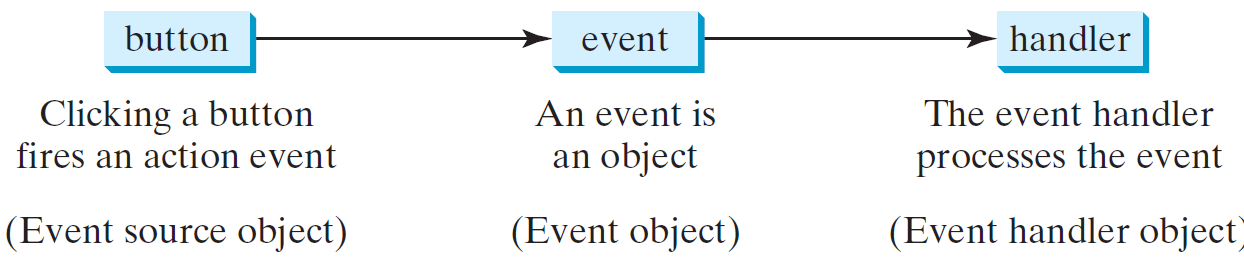
Chacune de ces action est appelée **évènement**. Dans **une programmation évènementielle**, le programme répond aux évènements générés. L’ordre des évènements est déterminé par l’utilisateur, pas par le programme (programmation procédurale). Chaque évènement généré, déclenche un traitement (méthodes) approprié.



Une GUI comporte 3 parties :

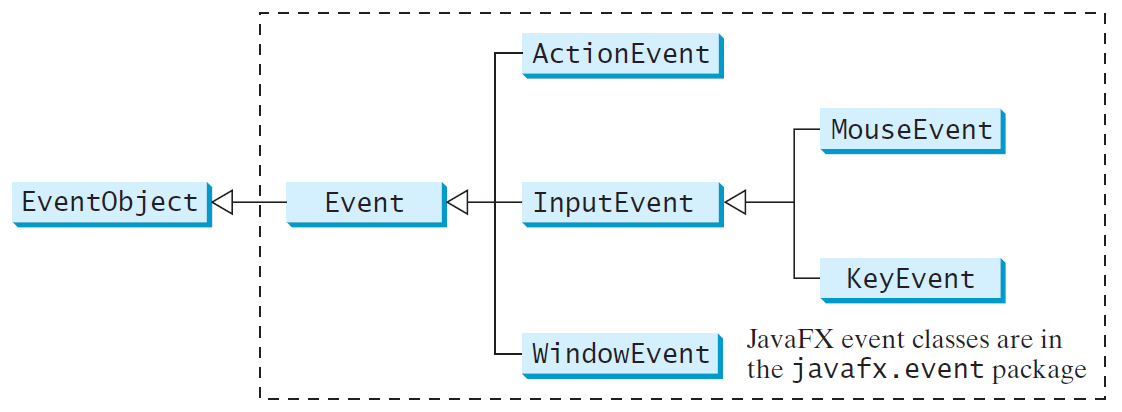
1. **Les composants graphiques** qui représentent la partie visible de l’interface (objets JavaFX tels que les controls, panes, shapes etc.) affichés à l’utilisateur et qui sont les sources des évènements.
2. **Les objets de type event** (ou évènements) qui sont des objets particuliers générées en actionnant les composants de l’interface.
3. **Des objets de type Handler** (traitement de l’évènement) qui reçoivent les évènements et y répondent en exécutant des actions/méthodes selon les besoins de l’application.

Par exemple, pour un bouton on a :



Pour capturer et traiter les évènements, les objets graphiques sont ˝écoutés˝ par des ˝écouteurs d’évènements˝ (Event Listeners) qui ont pour rôle de renvoyer l’évènement généré à l’objet qui va le traiter (Event handler).

Les différentes classes d’évènements se trouvent dans le package javaFX.event et sont organisés comme suit : :



Pour pouvoir répondre à un évènements, un programme doit:

* **créer** un objet écouteur d’évènement (**Event Listener)** spécifique pour cet évènement.
* Doit **insrire** l’objet écouteur avec le **composant** **graphique** qui a généré l’évènement
* Doit prévoir la **méthode appropriée** pour répondre à l’évènement déclenché

1. **principales étapes en programmation évènementielle**

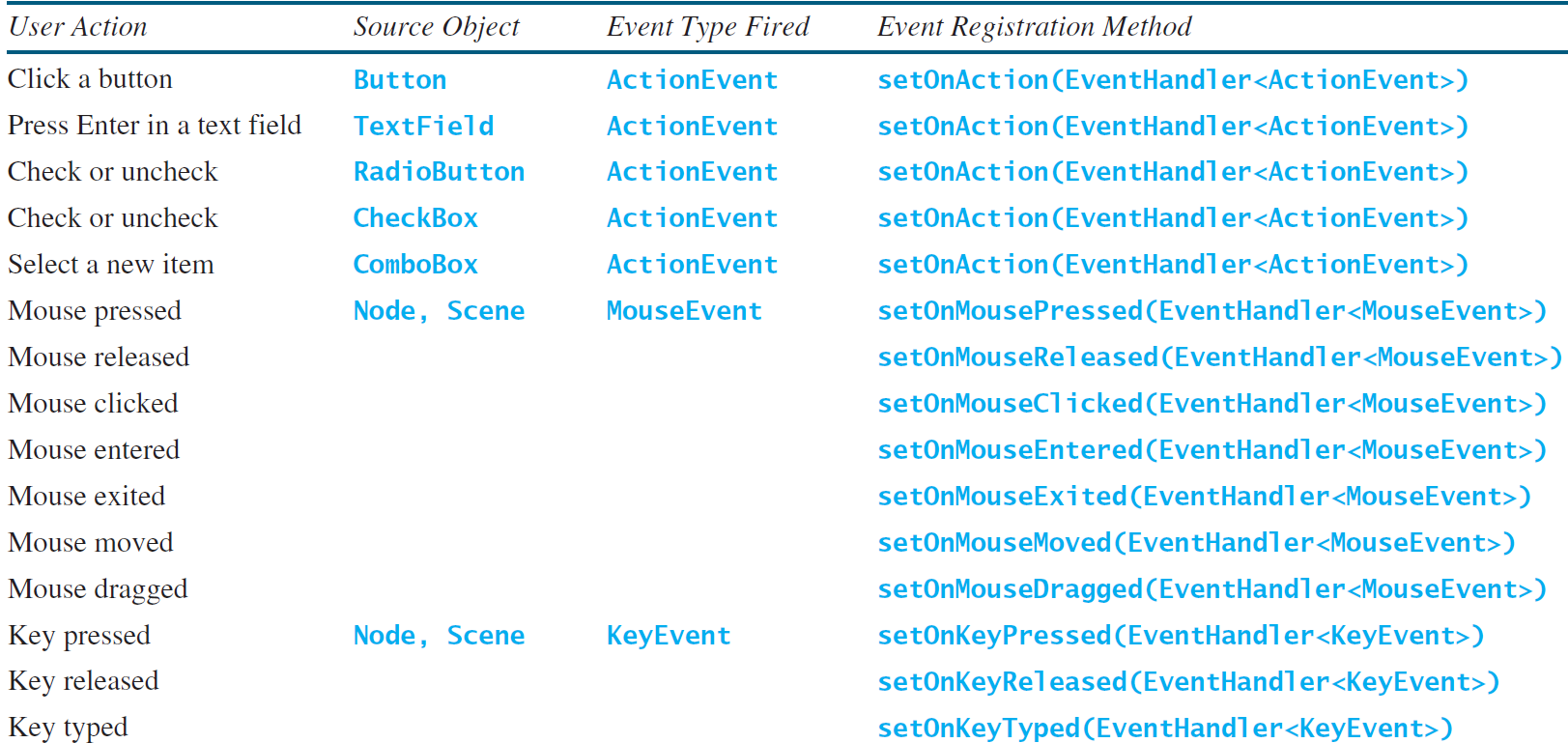
La programmation des évènements d’une interface doit passer par les étapes suivantes :

1. Identifier les catégories d’évènements déclenchés
2. Inscrire un « écouteur » pour chaque catégorie d’évènement
3. Ecrire les gestionnaires des évènements dans les classes des écouteurs

Nous allons expliquer ces 3 étapes en détails :

1. **1ere étape : Identifier les catégories d’évènements déclenchés**

Il y-a plusieurs types d’évènements parmi lesquels on a : **ActionEvent, MouseEvent** et **KeyEvent**

**Tableau 1 :** *Actions utilisateur, Objets Sources, évènements générés et types de traitements (Handler) correspondants*

A chaque catégorie d’évènements correspond le nom d’une méthode qui permet de lui associer un écouteur, formée par : **setOn** + nom de l’évènement (voir tableau1)

Exemple : à l’évènement **ActionEvent** Correspond la méthode **setOnAction**

à l’évènement **MouseEvent** Correspondent les méthodes **setOnMousexxxxx**

1. **2eme étape : Inscrire un « écouteur » sur chaque objet générant des évènements**

Pour cela, il suffit d’écrire :

***NomComposant.setOnxxxxx(EventHandler<yyyyyy>)***

Où :

* *NomComposant* : désigne l’objet composant graphique produisant l’évènement
* ***xxxxx*** désigne la catégorie d’évènement (voir tableaux), ex : Action, Mouse…
* Le paramètre ***EventHandler<yyyyyy>*** est une instance d’une classe qui implémente l’interface ***EventHandler<yyyyyy>***
* ***yyyyy*** est le type d’évènement généré.

1. **Ecrire les gestionnaires des évènements dans les classes des écouteurs**

* Chaque type d’évènement correspond généralement à un type d’interface que l’écouteur **doit implémenter**
* Chaque interface contient des signatures de méthodes, que l’écouteur va implémenter (remplir par son propre code). **Toutes les méthodes de l’interface doivent être implémentées**.

Pour implémenter les méthodes de l’interface, la classe de l’écouteur doit déclarer dans son en-tête : ***implements nom\_interface***

Pour répondre aux évènements, on utilise plusieurs alternatives :

* Par une classe interne à la classe dessinant l’interface (cas utilisé dans ce TP)
* Par une classe externe (voir TP3)
* Par une classe anonyme (voir TP3)

**Exercice 1**: dans cet exemple, nous utiliserons 2 classes internes : **OKHandlerClass** et **CancelHandlerClass** pour écouter les évènements associés aux boutons OK et Cancel.

**import javafx.application.Application;**

**import javafx.geometry.Pos;**

**import javafx.scene.Scene;**

**import javafx.scene.control.Button;**

**import javafx.scene.layout.HBox;**

**import javafx.stage.Stage;**

**import javafx.event.ActionEvent;**

**import javafx.event.EventHandler;**

**public class HandleEvent extends Application {**

**public void start(Stage primaryStage) {**

**// Create a pane and set its properties**

**HBox pane = new HBox(10);**

**pane.setAlignment(Pos.CENTER);**

**Button btOK = new Button("OK");**

**Button btCancel = new Button("Cancel");**

**OKHandlerClass handler1 = new OKHandlerClass();**

**btOK.setOnAction(handler1);**

**CancelHandlerClass handler2 = new CancelHandlerClass();**

**btCancel.setOnAction(handler2);**

**pane.getChildren().addAll(btOK, btCancel);**

**// Create a scene and place it in the stage**

**Scene scene = new Scene(pane);**

**primaryStage.setTitle("HandleEvent"); // Set the stage title**

**primaryStage.setScene(scene); // Place the scene in the stage**

**primaryStage.show(); // Display the stage**

**}**

**public static void main(String[] args) {**

**launch(args);**

**}**

**}**

**class OKHandlerClass implements EventHandler<ActionEvent> {**

**@Override**

**public void handle(ActionEvent e) {**

**System.out.println("OK button clicked");**

**}**

**}**

**class CancelHandlerClass implements EventHandler<ActionEvent> {**

**@Override**

**public void handle(ActionEvent e) {**

**System.out.println("Cancel button clicked");**

**}**

**}}**

Cet exemple permet d’afficher des messages sur la console en fonction du bouton cliqué.

**Exercice 2**: de la même manière, nous utilisons ici 2 boutons pour afficher une fenêtre contenant un cercle qui peut être agrandi en appuyant sur le bouton Enlarge ou rétréci en appuyant sur le bouton Shrink.

**import javafx.application.Application;**

**import javafx.event.ActionEvent;**

**import javafx.event.EventHandler;**

**import javafx.geometry.Pos;**

**import javafx.scene.Scene;**

**import javafx.scene.control.Button;**

**import javafx.scene.layout.StackPane;**

**import javafx.scene.layout.HBox;**

**import javafx.scene.layout.BorderPane;**

**import javafx.scene.paint.Color;**

**import javafx.scene.shape.Circle;**

**import javafx.stage.Stage;**

**public class ControlCircle extends Application {**

**private CirclePane circlePane = new CirclePane();**

**public void start(Stage primaryStage) {**

**// Hold two buttons in an HBox**

**HBox hBox = new HBox();**

**hBox.setSpacing(10);**

**hBox.setAlignment(Pos.CENTER);**

**Button btEnlarge = new Button("Enlarge");**

**Button btShrink = new Button("Shrink");**

**hBox.getChildren().add(btEnlarge);**

**hBox.getChildren().add(btShrink);**

**// Create and register the handler**

**btEnlarge.setOnAction(new EnlargeHandler());**

**btShrink.setOnAction(new ShrinkHandler());**

**BorderPane borderPane = new BorderPane();**

**borderPane.setCenter(circlePane);**

**borderPane.setBottom(hBox);**

**BorderPane.setAlignment(hBox, Pos.CENTER);**

**// Create a scene and place it in the stage**

**Scene scene = new Scene(borderPane, 400, 250);**

**primaryStage.setTitle("ControlCircle"); // Set the stage title**

**primaryStage.setScene(scene); // Place the scene in the stage**

**primaryStage.show(); // Display the stage**

**}**

**class EnlargeHandler implements EventHandler<ActionEvent> {**

**@Override // Override the handle method**

**public void handle(ActionEvent e) {**

**circlePane.enlarge();**

**}**

**}**

**class ShrinkHandler implements EventHandler<ActionEvent> {**

**@Override // Override the handle method**

**public void handle(ActionEvent e) {**

**circlePane.shrink();**

**}**

**}**

**public static void main(String[] args) {**

**launch(args);**

**}**

**}**

**class CirclePane extends StackPane {**

**private Circle circle = new Circle(50);**

**public CirclePane() {**

**getChildren().add(circle);**

**circle.setStroke(Color.BLACK);**

**circle.setFill(Color.WHITE);**

**}**

**public void enlarge() {**

**circle.setRadius(circle.getRadius() + 2);**

**}**

**public void shrink() {**

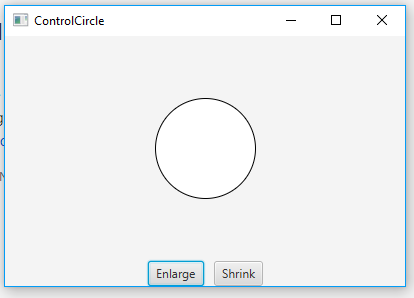
**circle.setRadius(circle.getRadius() > 2 ?**

**circle.getRadius() - 2 : circle.getRadius());**

**}**

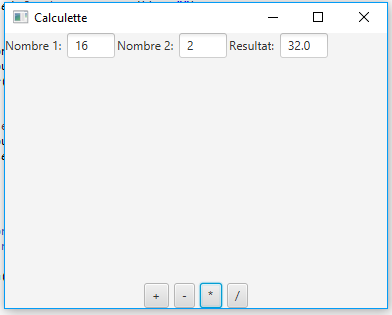
**}**

On obtient l’interface suivante :

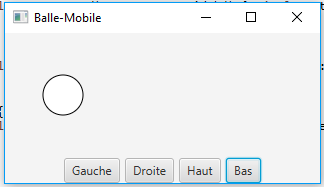


**Exercices d’application :**

1. **Calculatrice simple** : Ecrire un programme JavaFX permettant d’obtenir une calculatrice simple avec l’interface suivante :



1. La balle mobile : Ecrire un programme JavaFX permettant d’obtenir l’interface suivante :



En cliquant sur les boutons, la balle est déplacée de 2 pixels (en haut, en bas, à gauche ou à droite) de sa position actuelle.