Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

ATL – Ateliers Logiciels

Interfaces graphiques

Gestion des évènements

> **Consignes**
>
> Dans ce document vous trouverez les explications nécessaires à la gestion des actions et évènements via *JavaFX*. Il sera aussi l'occasion d'exposer deux nouveaux composants comme les menus et les boîtes de dialogue. Vous pouvez télécharger les ressources du TD via le lien `https://git.esi-bru.be/ATL/javaFX-Event/repository/archive.zip`

1 Évènement, action et gestionnaire d'évènement

Les composants disposés sur un écran, il faut maintenant s'interroger sur la manière dont l'utilisateur va interagir avec ceux-ci. Par exemple la classe `PrintText` (voir code 3 page 3) offre un écran qui permet d'ajouter au sein d'une zone prédéfinie le texte proposé par l'utilisateur.

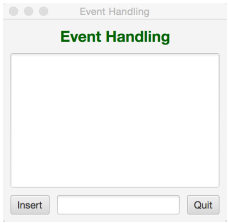

FIGURE 1 – Si l'utilisateur appuie sur le bouton `Insert`, on souhaite que le texte entré dans le composant `TextField` apparaisse au sein du composant `TextArea`.

Si l'utilisateur clique sur le bouton `Insert`, un évènement doit être déclenché.

Un évènement constitue une notification qui signale que quelque chose s'est passé. Il peut être provoqué par une action de l'utilisateur, comme un clic avec la souris, une pression sur une touche du clavier ou le déplacement d'une fenêtre mais aussi par la mise à jour d'un attribut, un *timer* arrivé à échéance, une information arrivée par le réseau...

En *JavaFX* les évènements sont représentés par des objets de la classe `Event` [1] .

Chaque instance de `Event` contient les informations suivantes :

- `EventType` : un évènement est d'un certain type. L'ensemble de ces types forment une hiérarchie. Si une touche du clavier est pressée, un évènement de type `KeyEvent.KEY_PRESSED` est généré. Ce type a pour parent le type `KeyEvent.ANY` qui englobe les différents évènements liés au clavier (`KeyEvent.KEY_RELEASED`, `KeyEvent.KEY_TYPED`,...) ;

1. `https://openjfx.io/javadoc/13/javafx.base/javafx/event/Event.html`

Version du document
29 septembre 2021

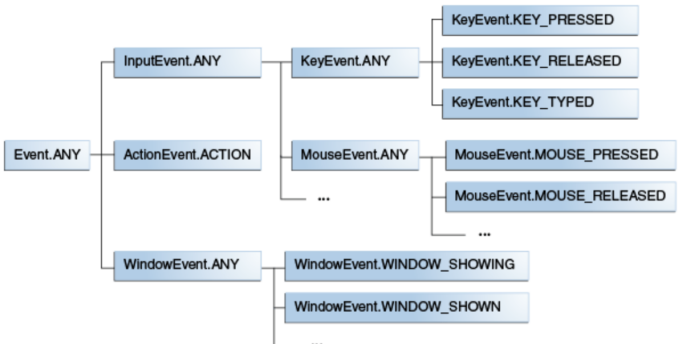

FIGURE 2 – Hiérarchie des types d'évènements.

- ▷ la source : il s'agit de l'objet qui est à l'origine de l'évènement ;
- ▷ la cible (`Target`) : la cible de l'évènement.

Une fois généré, cet évènement va se propager le long du chemin correspondant à la chaine de traitement (`Event Dispatch Chain`), depuis la racine (`Stage`) jusqu'à la cible (`Target`). L'évènement pourra alors être arrêté via des filtres (`Event Filter`) associés aux nœuds qu'il parcourt.

Si aucun filtre ne l'arrête, l'évènement remonte ensuite depuis la cible jusqu'à la racine et les gestionnaires d'évènement (`Event Listener`) sont exécutés dans l'ordre de passage. A moins qu'il ne soit consommé, un évènement dont le traitement est terminé par un nœud est passé au nœud suivant de la chaine de traitement.

Le traitement des évènements comporte les étapes suivantes :

- ▷ La sélection de la cible de l'évènement. Si plusieurs composants se trouvent à un emplacement donné c'est celui qui est "au-dessus" qui est considéré comme la cible ;
- ▷ La détermination de la chaine de traitement des évènements (*Event Dispatch Chain*). Ce chemin part de la racine (*Stage*) et va jusqu'au composant cible en parcourant tous les nœuds intermédiaires ;
- ▷ Le traitement des filtres d'évènement (*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'évènement (*Event Handler*). Exécute le code des gestionnaires d'évènement en suivant le chemin montant, du composant cible à la racine (*Stage*) ;
- ▷ Si un filtre ou un gestionnaire d'évènement consomme l'évènement, celui-ci ne sera pas propagé au noeud suivant de la chaine de traitement.

Si nous souhaitons que le bouton `Insert` de la classe `PrintText` imprime le texte encodé à l'écran, plusieurs solutions s'offrent à nous.

1.1 Gestionnaire d'évènement : le handler

Nous pouvons créer la classe `InsertButtonHandler` qui servira de contrôleur. Cette classe implémente l'interface `EventHandler` et effectue les opérations souhaitées via la méthode `handle()`.

```java
public class PrintText
      extends Application {

   public static void main(String[] args) {
      launch(args);
   }

   private final BorderPane root = new BorderPane();

   private final HBox btnPanel = new HBox(10);
   private final Label lblTitle = new Label("Event Handling");
   private final TextArea txaMsg = new TextArea();
   private final Button btnInsert = new Button("Insert");
   private final TextField tfdCharacter = new TextField();
   private final Button btnQuit = new Button("Quit");
   @Override
   public void start(Stage primaryStage) throws Exception {

      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);
      //--- Button Panel
      btnPanel.getChildren().add(btnInsert);
      btnPanel.getChildren().add(tfdCharacter);
      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();

   }
}
```

FIGURE 3 – Classe PrintText

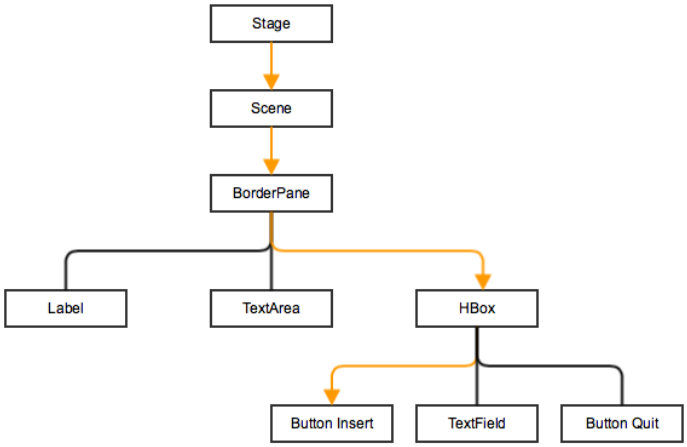

FIGURE 4 – Lorsque l'utilisateur appuie sur le bouton `Insert` de la classe `PrintText` l'évènement va se propager dans le graphe de scène en partant de la racine du graphe, le `Stage`, vers la cible de l'évènement, le bouton `Insert`.

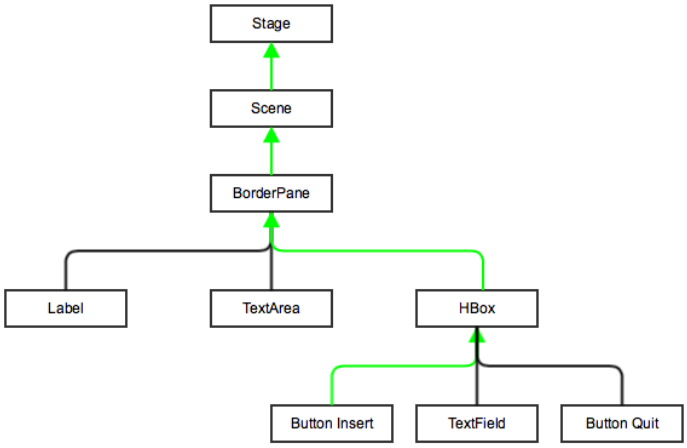

FIGURE 5 – Si aucun filtre ne l'arrête, l'évènement, à moins d'être consommé par un `event handler`, remonte ensuite le graphe de scène de la classe `PrintText` depuis la cible, le bouton `Insert`, jusqu'à la racine, le `Stage`.

```java
public class InsertButtonHandler
      implements EventHandler<ActionEvent> {

   private final TextArea tArea;
   private final TextField tfdText;
   //--- Constructeur ----------------------------------
   public InsertButtonHandler(TextArea tArea, TextField tfdText) {
      this.tArea = tArea;
      this.tfdText = tfdText;
   }

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

Remarquez que le constructeur du gestionnaire d'évènement reçoit en paramètres les composants impliqués lors du traitement de l'évènement, dans notre cas il s'agit des composants `TextArea` et `TextField`.

Au sein de la classe `PrintText` il faut créer une instance de `InsertButtonHandler` et l'enregistrer comme gestionnaire d'évènement sur le bouton `Insert` en invoquant la méthode `addEventHandler()`.

```
1    //--- Button Events Handling
2    InsertButtonHandler insertCtrl = new InsertButtonHandler(txaMsg,tfdCharacter);
3    btnInsert.addEventHandler(ActionEvent.ACTION, insertCtrl);
```

La méthode `addEventHandler`[2] agit sur une instance de `Node` et prend en paramètres le type d'évènement auquel il doit réagir ainsi que le gestionnaire de cet évènement.

1.2 Gestionnaire d'évènement : classes imbriquées et classes anonymes

Jusqu'à présent, les classes que nous avons utilisées étaient toutes des classes de premier niveau, c'est-à-dire des membres directs des paquetages, sans imbrication. Le langage *Java* permet cependant de définir des classes à l'intérieur d'autres classes. Il s'agit de classes imbriquées ou de *Nested Class*. Elles sont classées en deux catégories :

Les deux types de classes imbriquées :

- Les classes imbriquées statiques (*Static Nested Class*)
- Les classes imbriquées non statiques (*Inner Class*)
  - Les classes locales (*Local Inner Class*)
  - Les classes anonymes (*Anonymous Inner Class*)

Dans le cadre de ce TD, nous utiliserons uniquement les classes anonymes dont voici le tutoriel Oracle[3] . Nous vous invitons cependant à consulter le tutoriel concernant l'ensemble des types de classes internes[4] .

Nous avons vu que chaque nœud possède une méthode `addEventHandler()` permettant d'ajouter l'instance d'un gestionnaire d'évènement. Ce gestionnaire est parfois uniquement utilisé pour ce composant précis et la méthode `handle()` utilisée n'est pas souvent très longue. Il semble alors contraignant de dédier une nouvelle classe pour ce gestionnaire. Dans ce cas, passons-nous de la création de notre classe `InsertButtonHandler` et utilisons une classe sans nom, dite anonyme.

```
1    //--- Button Events Handling
2    btnInsert.addEventHandler(ActionEvent.ACTION, new
     ↪ EventHandler<ActionEvent>() {
3
4        @Override
5        public void handle(ActionEvent event) {
6            txaMsg.appendText(tfdCharacter.getText());
7        }
8    });
```

Une classe anonyme qui implémente une interface ne contenant qu'une seule méthode peut s'écrire avec une *lambda*.

2. https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/Node.html
3. https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html
4. https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

```
//--- Button Events Handling
btnInsert.addEventHandler(ActionEvent.ACTION, e -> {
      txaMsg.appendText(tfdCharacter.getText());
});

//--- Button Events Handling
btnInsert.addEventHandler(
   ActionEvent.ACTION,
   e -> txaMsg.appendText(tfdCharacter.getText())
);
```

1.3 Gestionnaire d'évènement : méthode `setOn`*`EventType`*

La plupart des composants possèdent des méthodes dédiées à l'enregistrement d'un gestionnaire particulier. Par exemple, comme le montre l'exemple ci-dessous, pour enregistrer un gestionnaire sur le bouton `Insert` nous pouvons appeler la méthode `setOn`*`Action`*`()` et lui passer en paramètre une instance de classe anonyme qui implémente l'interface `EventHandler`.

```
//--- Button Events Handling
btnInsert.setOnAction(new EventHandler<ActionEvent>() {

   @Override
   public void handle(ActionEvent event) {
      txaMsg.appendText(tfdCharacter.getText());
   }
});
```

Version *lambda*

```
//--- Button Events Handling
btnInsert.setOnAction(e -> txaMsg.appendText(tfdCharacter.getText()));
```

Pour chaque composant utilisé, il faut faire l'inventaire des méthodes `setOn`*`EventType`* disponibles. Ci-dessous, vous trouverez une liste des actions associées à des méthodes `setOn`*`EventType`* les plus courantes :

Composant	Evènement	Méthode
`Node`	`KeyEvent`	Pression sur une touche du clavier
`Node`	`MouseEvent`	Déplacement de la souris ou pression sur un de ses boutons
`Node`	`MouseDragEvent`	Glisser-déposer avec la souris
`Node`	`ScrollEvent`	Composant scrollé
`ButtonBase`	`ActionEvent`	Bouton cliqué
`ComboBoxBase`	`ActionEvent`	`ComboBox` ouverte ou fermée
`ContextMenu`	`ActionEvent`	Une des options d'un menu contextuel activée
`MenuItem`	`ActionEvent`	Option de menu activée
`TextField`	`ActionEvent`	Pression sur Enter dans un champ texte
`Menu`	`Event`	Menu est affiché ou masqué
`Window`	`WindowEvent`	Fenêtre affichée, fermée, masquée

1.4 Exercice

En utilisant le gestionnaire d'évènement `GraphDisplayHandler` ci-dessous :

```
public class GraphDisplayHandler
      implements EventHandler<MouseEvent> {

   private final String text;

   public GraphDisplayHandler(String component) {
      this.text = component;
   }

   @Override
   public void handle(MouseEvent event) {
      System.out.println(text);
   }
}
```

Modifions la classe `PrintText` pour obtenir la classe `PrintTextGraphDisplay` en y insérant le code suivant

```
      //--- Button Events Handling
      btnInsert.setOnAction(new EventHandler<ActionEvent>() {

         @Override
         public void handle(ActionEvent event) {
            txaMsg.appendText(tfdText.getText());
         }
      });

      primaryStage.addEventFilter(MouseEvent.MOUSE_CLICKED,
                           new GraphDisplayHandler("Stage Filter"));

      primaryStage.addEventHandler(MouseEvent.MOUSE_CLICKED,
                            new GraphDisplayHandler("Stage Handler"));

      scene.addEventFilter(MouseEvent.MOUSE_CLICKED,
                     new GraphDisplayHandler("Scene Filter"));

      scene.addEventHandler(MouseEvent.MOUSE_CLICKED,
                      new GraphDisplayHandler("Scene Handler"));

      root.addEventFilter(MouseEvent.MOUSE_CLICKED,
                    new GraphDisplayHandler("BorderPane Filter"));

      root.addEventHandler(MouseEvent.MOUSE_CLICKED,
                     new GraphDisplayHandler("BorderPane Handler"));

      btnInsert.addEventFilter(MouseEvent.MOUSE_CLICKED,
                        new GraphDisplayHandler("Insert Button Filter"));

      btnInsert.addEventHandler(MouseEvent.MOUSE_CLICKED,
                         new GraphDisplayHandler("Insert Button Handler"));

      btnPanel.addEventFilter(MouseEvent.MOUSE_CLICKED,
                        new GraphDisplayHandler("HBox Filter"));

      btnPanel.addEventHandler(MouseEvent.MOUSE_CLICKED,
                         new GraphDisplayHandler("HBox Handler"));
```

Question 1

Exécutez la classe `PrintTextGraphDisplay` et insérez un texte dans le `TextArea` via le bouton `Insert`. Du texte apparaît maintenant dans la console. Comment interprétez-vous ce résultat ? N'hésitez pas à vous aider des figures 4 page 4 et 5 page 4.

1.5 La méthode `consume()`

Avant d'arriver jusqu'aux gestionnaires, un évènement doit passer au travers des différents filtres présents dans la chaine de traitement. Ceux-ci peuvent avoir comme effet de consommer l'évènement et de l'empêcher de passer au nœud suivant.

Question 2

1. Dans la classe `PrintText` ajoutez le code ci-dessous au composant `TextField` :

```
tfdText.addEventFilter(
   KeyEvent.KEY_TYPED,
   e -> event.consume()
);
```

   Quel est l'effet de ce filtre ?

2. Comment pouvez-vous modifier le filtre pour qu'il consomme uniquement l'encodage des caractères et laisse passer l'encodage de chiffres ?

1.6 Multiplicité des filtres et des gestionnaires

Si un nœud du graphe de scène possède plusieurs récepteurs d'évènements enregistrés, l'ordre d'activation de ces récepteurs sera basé sur la hiérarchie des types d'évènement :

- ▷ Un récepteur pour un type spécifique sera toujours exécuté avant un récepteur pour un type plus générique. Un filtre enregistré pour `MouseEvent.MOUSE_PRESSED` sera exécuté avant un filtre pour `MouseEvent.ANY` qui sera exécuté avant un filtre pour `InputEvent.ANY` ;

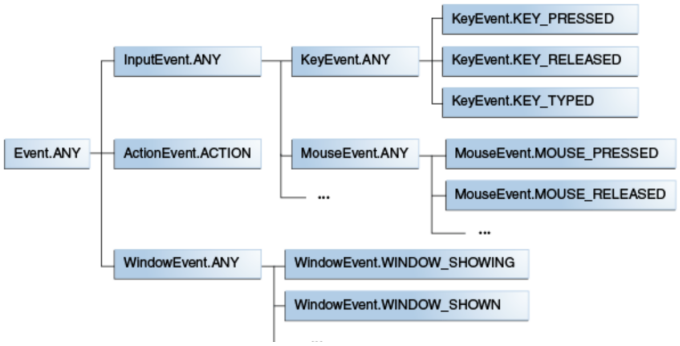

FIGURE 6 – Hiérarchie des types d'évènements

- ▷ L'ordre d'exécution des récepteurs pour des types de même niveau n'est pas défini.

2 Menu

Les menus sont des éléments de l'interface permettant à l'utilisateur de choisir des options qui pourront déclencher des actions et/ou changer l'état de certaines propriétés de l'application. Le principe du menu est que l'utilisateur puisse voir et parcourir la liste des options avant de se décider. Dans les applications, les menus peuvent prendre différentes formes. Parmi les plus classiques, on trouve :

- Les menu déroulants (*drop-down menu*) : des en-têtes de menus sont placés dans un conteneur sous forme de barre. Un clic sur ces en-têtes « déroule » le menu et fait apparaitre, dans une fenêtre *popup*, les options de ce menu.
- Les menu contextuels (*popup menu*) : menus affichés en réaction à un évènement, généralement un clic droit de la souris. Les options du menu qui sont affichées dépendent du contexte, c'est-à-dire de l'endroit où l'on a cliqué.

Les options des menus peuvent elles-mêmes ouvrir d'autres menus. On parle alors de sous-menus qui peuvent s'ouvrir en cascade.

Dans la librairie *JavaFX*, un certain nombre de composants sont dédiés aux menus et sont utilisés pour les construire :

- `MenuBar`
- `Menu`
- `MenuItem`
- `CheckMenuItem`
- `RadioMenuItem`
- `CustomMenuItem`
- `SeparatorMenuItem`
- `Contextmenu`

Afin d'apprendre la syntaxe nécessaire à l'utilisation de menus nous vous demandons de suivre le tutoriel Oracle [5] sur le sujet.

3 Boite de dialogue

Les boites de dialogue sont des éléments d'une interface graphique qui se présentent généralement sous la forme d'une fenêtre affichée par une application dans le but d'informer l'utilisateur, d'obtenir une information de l'utilisateur ou une combinaison des deux. Une boite de dialogue peut être :

- modale : l'utilisateur ne peut pas interagir avec la fenêtre dont la boite de dialogue dépend avant de l'avoir fermée ;
- non-modale : l'utilisateur peut interagir avec la boite de dialogue mais aussi avec la fenêtre dont la boite de dialogue dépend.

La plupart des kits de développement offrent des solutions clés en main pour les boites de dialogue simples. *JavaFX* nous propose les différents types de boites de dialogue suivantes par défaut :

- `Alert` [6] : boite d'informations, de warning, d'erreur ou de confirmation (voir table 1 page suivante) ;

5. http://docs.oracle.com/javafx/2/ui_controls/menu_controls.htm
6. https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/Alert.html

TABLE 1 – Différents types de boites d'alertes proposées par *JavaFX*

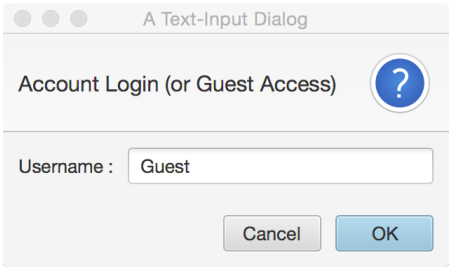

FIGURE 7 – le composant `TextInputDialog` permet de recueillir des informations auprès de l'utilisateur.

- `TextInputDialog`[7] : pour saisir une ligne de texte (voir figure 7) ;
- `ChoiceDialog`[8] : pour saisir une ligne de texte (voir figure 8).

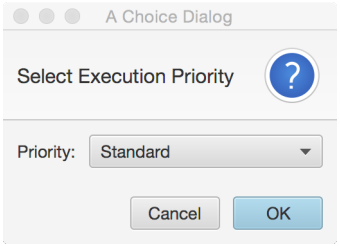

FIGURE 8 – Le composant `ChoiceDialog` permet de proposer un choix à l'utilisateur.

7. https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/TextInputDialog.html
8. https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/ChoiceDialog.html

3.1 Boite de dialogue particulière

Il est toutefois possible de personnaliser ces boites de dialogue en utilisant directement la classe `Dialog` qui est la classe parente des classes `Alert`, `TextInputDialog` et `ChoiceDialog`.

On peut également utiliser des boites de dialogues dédiées à des usages plus précis :

- `FileChooser`[9] : permet à l'utilisateur de naviguer dans l'arborescence des fichiers de la machine cible et de sélectionner un ou plusieurs fichiers ;
- `DirectoryChooser`[10] : permet à l'utilisateur de naviguer dans l'arborescence des fichiers de la machine cible et de sélectionner un répertoire ;
- `DatePicker`[11] : permet à l'utilisateur de sélectionner une date dans un calendrier qui est affiché en fenêtre *popup* ;
- `ColorPicker`[12] : permet à l'utilisateur de sélectionner une couleur dans une palette.

9. https://openjfx.io/javadoc/17/javafx.graphics/javafx/stage/FileChooser.html
10. https://openjfx.io/javadoc/17/javafx.graphics/javafx/stage/DirectoryChooser.html
11. https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/DatePicker.html
12. https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/ColorPicker.html