Développement d'application avec IHM

Gestion des événements

Christine Jacquin

La programmation événementielle

Le développement d'applications avec IHM est basée sur un paradigme de programmation nommé **programmation événementielle**

- **Programmation impérative (séquence d'instructions)**

  le programme est le chef d'orchestre (par exemple, il demande à l'utilisateur d'entrer des valeurs, calcule et affiche un résultat, etc.
- **Programmation événementielle**

  - ce sont les événements (généralement déclenchés par l'utilisateur, mais aussi par le système) qui pilotent l'application.

  - 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

Les événements

Un événement peut être provoqué par :

Une action de l'utilisateur

- Un clic avec la souris
- L’appui sur une touche du clavier
- Le déplacement d'une fenêtre
- Un geste sur un écran tactile
- ...

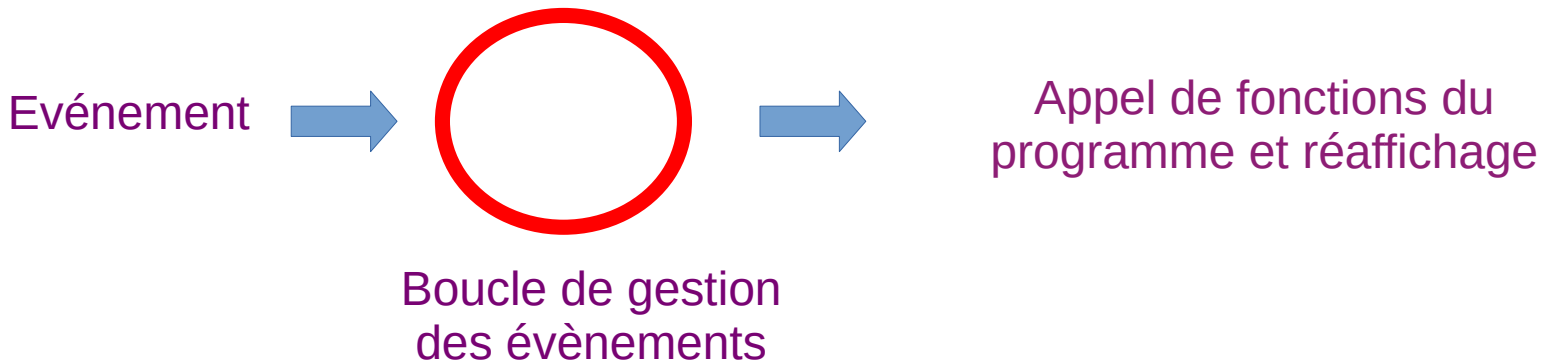
Un changement provoqué par le système

- Un timer est arrivé à échéance
- Un processus a terminé un calcul
- Une information est arrivée par le réseau
- ...

La boucle de gestion des événements

Boucle infinie qui :

- Récupère les événements
- Notifie les composants
- Lancée automatiquement à l'initialisation du programme



Qu’est-ce qu’un événement en javaFX ?

Un objet de la classe **Event** à partir duquel on peut connaître :

- **Le type** : par exemple, la classe **KeyEvent** qui correspond aux événements liés au clavier qui engloble *KEY_PRESSED, KEY_RELEASE, KEY_TYPED.* On y accède via l’attribut ***eventType***
- **La source** de l’événement => clavier, souris via l’attribut ***source***
- **La cible** d’un événement => élément (Node) sur lequel l’événement s’est produit (un bouton par exemple). On y accède via l’attribut ***target***

Le " stage " JavaFX reçoit les événements et les diffusent au composant cible

Gestion des événements

La gestion des événements suit les étapes suivantes (similaire à ce qu'il se passe en javascript):

- **La sélection par l'utilisateur de la cible (Target) de l'événement**

  Si plusieurs composants se trouvent à un emplacement donné, celui qui est "au-dessus" 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. L'objet stage propage l'événement sur le chemin construit.

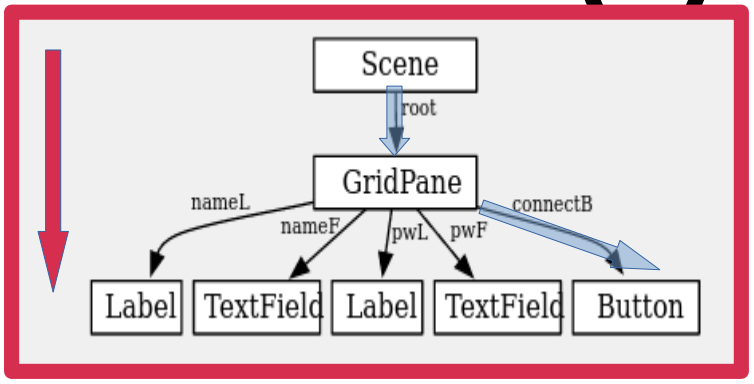
- **Le traitement des filtres d'événement (Event Filter)**

  Exécute le code des filtres en suivant le chemin descendant, de la racine (Stage) jusqu'au composant cible : **Event Capturing phase**

- **Le traitement des gestionnaires d'événement (Event Handler)**

  Exécute le code des gestionnaires d'événement en remontant le chemin, du composant cible à la racine (Stage) : **Event Bubbling phase.** La propagation est terminée dès lors que l'événement est pris en charge dans un noeud

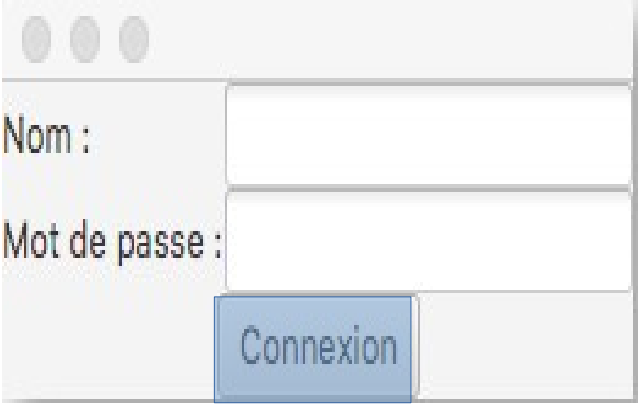
Exemple de gestion d'événement(1)



Première phase :

Lorsque le bouton est cliqué, la chaîne de traitement est générée par l'objet *Stage* et l'évènement est propagé dans l'arbre des nœuds jusqu'au nœud cible (*Button* ici)

Si un nœud traversé comporte un filtre (**Event Filter**) alors il est exécuté (dans l'ordre de passage). L'événement se propage jusqu'au nœud cible

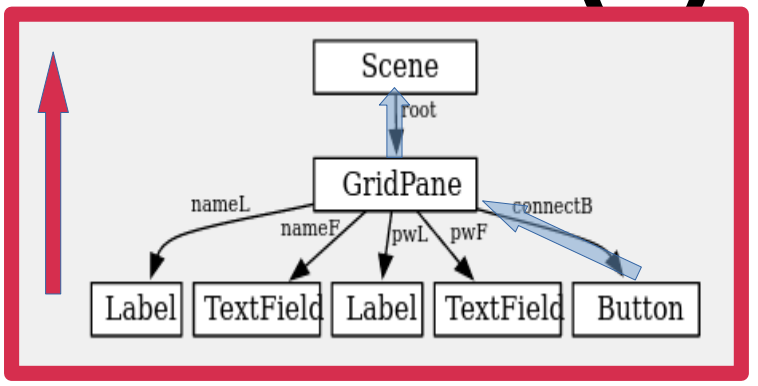


https://cs108.epfl.ch/archive/17/c/JAFX/JAFX-notes.html

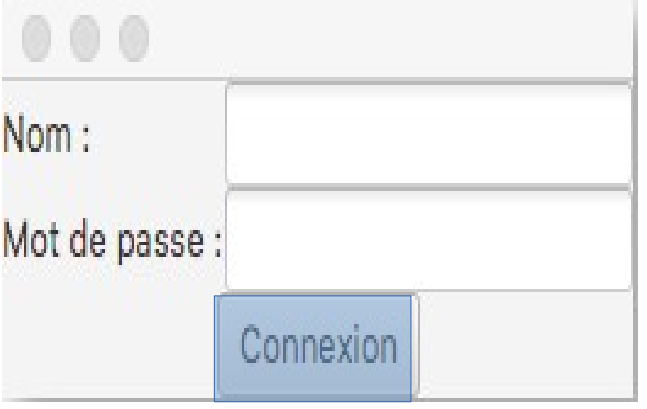
Exemple de gestion d’événement(2)

Deuxième phase :

L'événement remonte ensuite depuis la cible (Button) jusqu'à la racine (stage) et le premier nœud possédant un gestionnaire d'événement éligible (Event Listener) est exécuté. La propagation s’arrête alors.



Il est possible au niveau de la première phase (descente), d’arrêter la propagation, en utilisant la méthode *consume()* dans le filtre.



https://cs108.epfl.ch/archive/17/c/JAFX/JAFX-notes.html

Les gestionnaires d’évènement

Pour gérer un événement, il faut créer un **écouteur** d’évènement (Event Listener) et l'enregistrer sur les nœuds du graphe de scène où l'on souhaite intercepter l'événement et effectuer un traitement.

Un écouteur d'événement peut être enregistré comme **filtre** ou comme **gestionnaire d’événement** suivant le comportement attendu

Les filtres, comme les gestionnaires d'événements, sont des objets qui doivent implémenter l'interface :

`EventHandler<Event>` qui est composée de l'unique méthode :

`fun handle(event : Event)` qui se charge de traiter l'événement.

Ajout d’écouteur

Pour enregistrer un écouteur d'événement sur un nœud du graphe de scène, on peut utiliser soit:

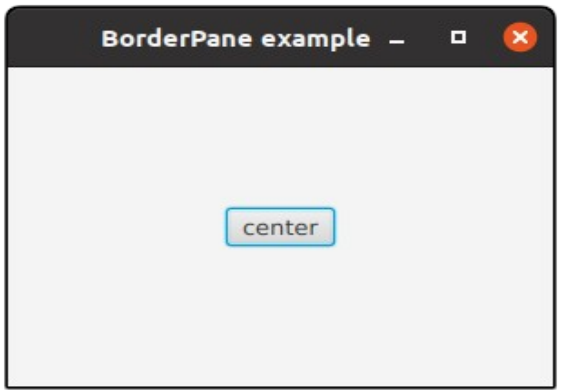
- la méthode **addEventFilter()** pour lui associer un filtre (Filter)
- la méthode **addEventHandler()** pour lui associer un gestionnaire d'événement

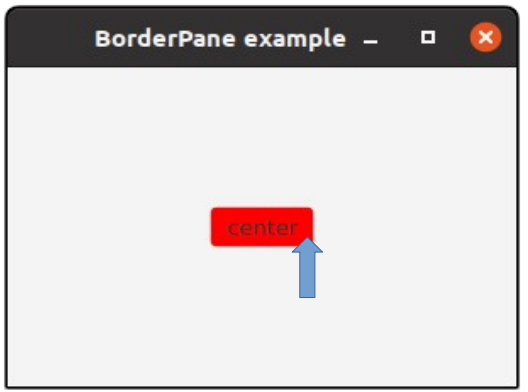
Ou pour les gestionnaires d’événement:

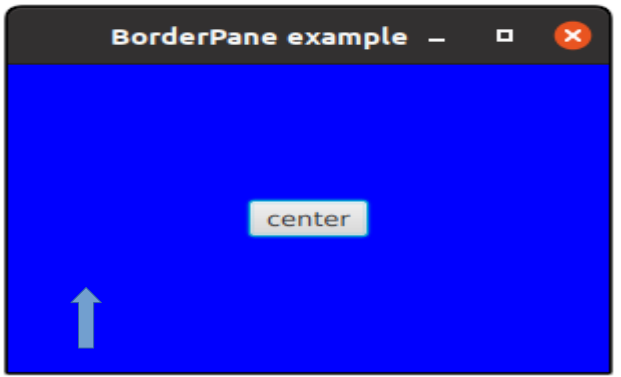
- Utiliser certaines méthodes spécifiques sur certains composants et qui permettent d'enregistrer un gestionnaire d'événement en tant que propriété du composant.

  Par exemple : **setOnAction(***fonctionHandler)* ou **setOnKeyUp(***fonctionHandler)*

Exemple de gestion d’événement (1)



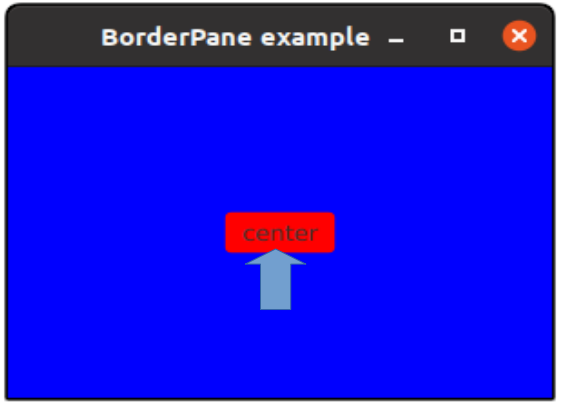




Des gestionnaires d’événement sont associés au *Button* et au *BorderPane* :

Button => au clic, il se colorie en rouge
BorderPane => au clic, il se colorie en bleu

Exemple de gestion d’événement (2)

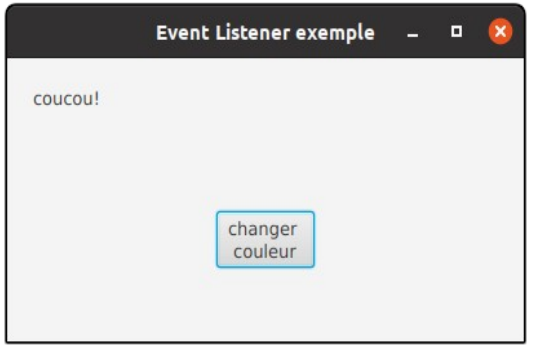


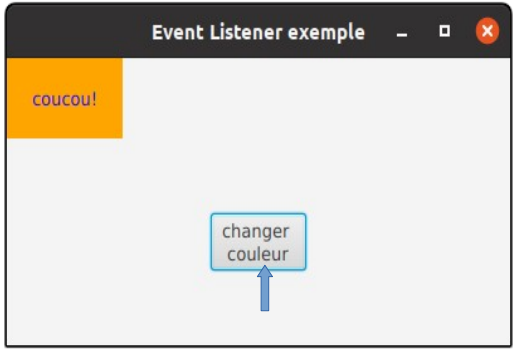
Maintenant, un filtre est associé au BorderPane qui au clic, colorie le panneau en bleu. Le bouton a toujours le même gestionnaire qu’à la diapo précédente.

Par le mécanisme de descente et de remontée, les 2 écouteurs sont déclenchés au clic sur le bouton

Création d’un EventListener

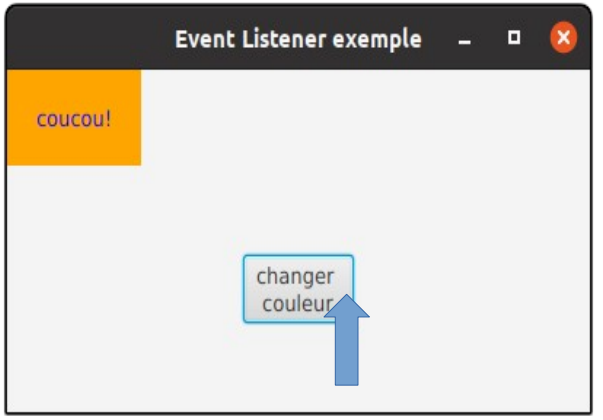
```
public class EcouteurBouton(Label label) : EventHandler<ActionEvent ?> {
private val label : Label
//Constructeur
init {
this.label=label
}

//Code exécuté lorsque l'événement survient
override fun handle(event : ActionEvent ?) {
this.label.style= "-fx-background-color: orange"
this.labell.textFill=Color.BLUE
}

}
```





Association à un composant

```
bouton=Button("changer \n couleur")
label=Label("coucou!")
ecouteurBouton=EcouteurBouton(label)
bouton.addEventHandler(ActionEvent.ACTION,ecouteurBouton)
```



Maintenant au clic sur le bouton, le label se coloriera

A noter que le premier paramètre de la méthode **addEventHandler()** renseigne le type d'événement que l'on écoute.

Autres syntaxes

Il existe des méthodes spécifiques aux composants pour leur associer des gestionnaires d'évènements : setOn*Evenement*(EventHandler<Event>)

```
monBouton.setOnAction(ecouteurBouton)
monTextfield.setOnKeyPressed(ecouteurMonTexte)
```

On peut passer aussi en paramètre, une lambda expression

```
monBouton.setOnAction{
label.style= "-fx-background-color: orange"
label.textFill=Color.BLUE
}
```

ou autre manière d'utiliser addEventHandler()

```
monBouton.addEventHandler(ActionEvent.ACTION,{
label.style= "-fx-background-color: orange"
labell.textFill=Color.BLUE
})
```

Utilisation des lambdas expressions

Les méthodes **setOn*Evenement*()** et **addEvenHandler()** peuvent prendre respectivement comme premier et second paramètre une **lambda expression**. Une fonction anonyme est passée en paramètre => raccourci syntaxique qui permet de définir une méthode directement à l'endroit où elle est utilisée

```
monBouton.setOnAction{
label.style= "-fx-background-color: orange"
}
```

Avantages :

- Diminue le nombre de classes
- Facile à implémenter avec un bon IDE
- Facilite la lecture du code

Privilégier l'utilisation seulement pour des traitements courts

Inconvénients

- Moins de réutilisation / factorisation de code possible
- Moins de séparation des préoccupations
- **Complexifie la lecture du code et ne facilite pas la maintenance**

Les différents événements (1)

Liste des principales actions associées aux méthodes setOn*Evenement*() qui permettent d'associer des gestionnaires d'événements aux diverscomposants

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

Les différents évènements (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`

Les fenêtres de dialogue

Les boîtes de dialogue sont des éléments d'une interface graphique qui se présentent généralement sous la forme d'une fenêtre affichée par une application (ou éventuellement par le système d'exploitation) dans le but :

- d'informer l'utilisateur (texte, mise en garde, ...)
- d'obtenir une information de l'utilisateur (mot de passe, choix, ...)
- ou une combinaison des deux

Une boîte de dialogue dépend d'une autre fenêtre

Boîte de dialogue modale / non modale

Modale

- L'utilisateur ne peut pas interagir avec la fenêtre dont la boîte de dialogue dépend avant de l'avoir fermée
- Une boîte de dialogue modale sera utilisée pour confirmer ou annuler une action critique (suppression de données par exemple).

- La fenêtre principale est bloquée tant que l'utilisateur n’a pas confirmé ou infirmé son choix
- La boîte de dialogue se ferme automatiquement dès la décision prise.

Non-modale

- L'utilisateur peut interagir avec la boîte de dialogue mais aussi avec la fenêtre dont la boîte de dialogue dépend (en laissant la boîte de dialogue ouverte).
- Une boîte de dialogue non-modale sera utilisée par exemple pour fournir à l'utilisateur une palette d'outils qu'il pourra sélectionner et appliquer sur la fenêtre principale.

La boîte de dialogue reste ouverte tant que l'utilisateur ne la ferme pas explicitement.

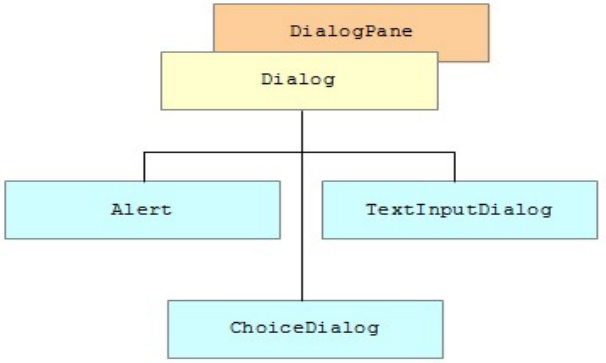
Boîte de dialogue et JavaFX

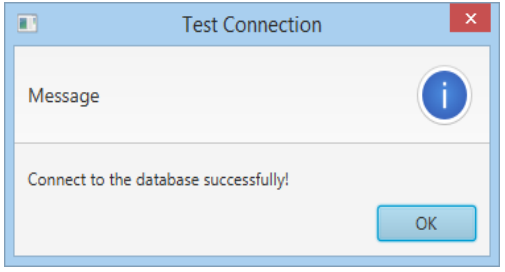
Il existe 3 types principaux de boîtes de dialogue :

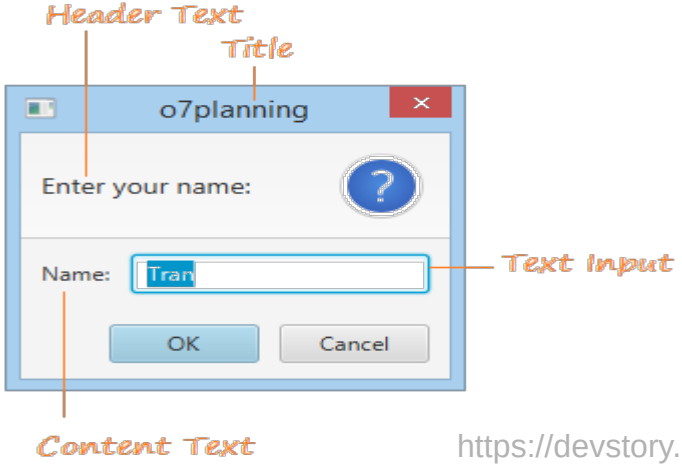
Alert : permet d'afficher un message à l'utilisateur

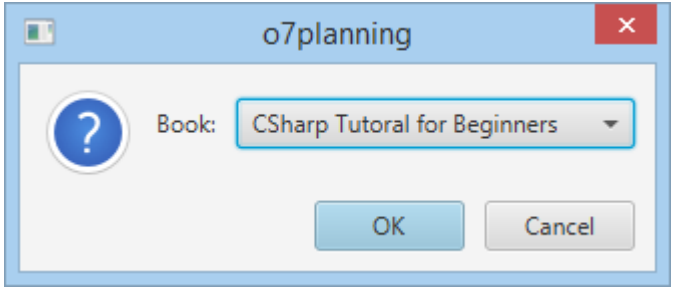
TextInputDialog : permet à l'utilisateur de saisir un texte

ChoiceDialog : permet à l'utilisateur de réaliser un choix



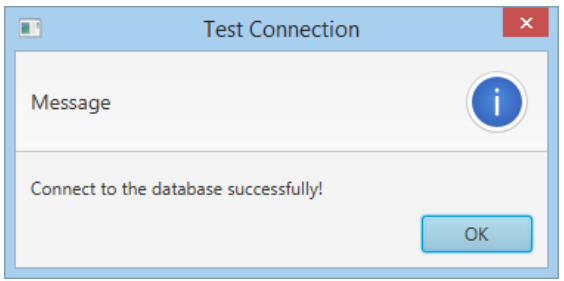


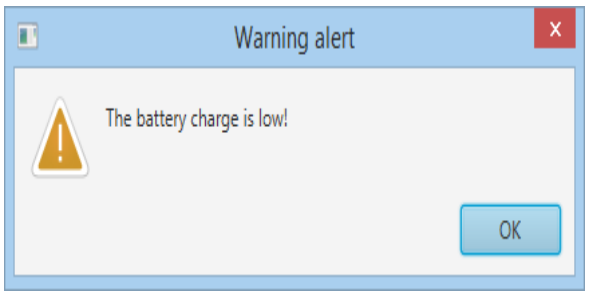


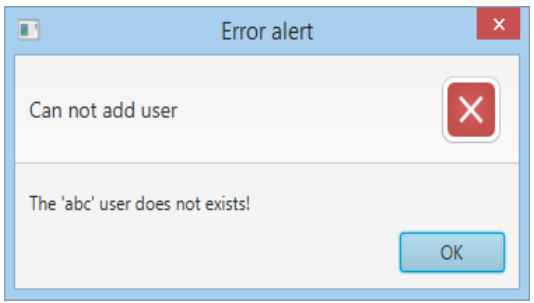


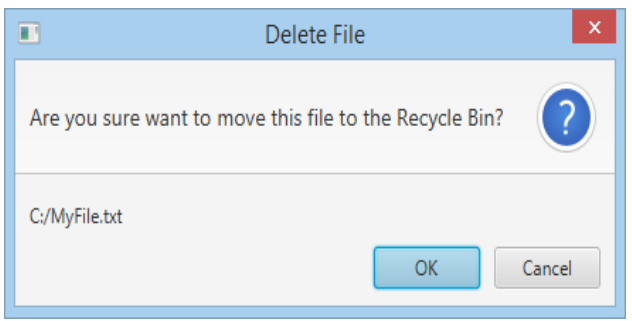
Exemple : Boîte de dialogue Alert

Plusieurs type de boîte : Information, Warning, Error, Confirmation



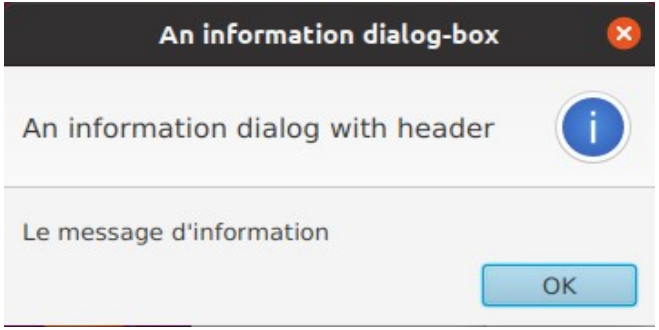






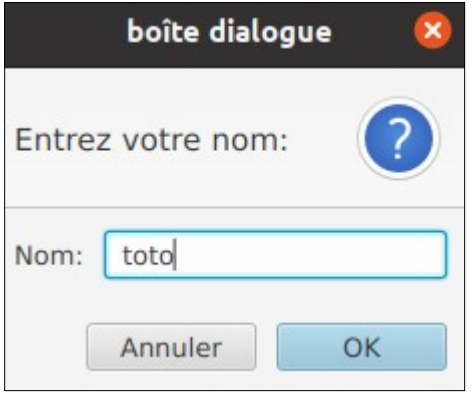
Exemple de code

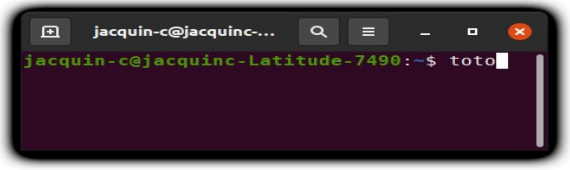
```
private fun showWarningAlert(){
        val dialog = Alert(AlertType.INFORMATION)
        dialog.title="An information dialog-box"
        dialog.headerText="An information dialog with header"
        dialog.contentText="Le message d'information "
        dialog.showAndWait()
}


override fun start(premierStage Stage) {
                …
                val root = BorderPane()
                val bouton= Button("Warning Alert")
                bouton.setOnAction{showWarningAlert}
                root.center=bouton
                …
}
```



Exemple de code pour TextInputDialog

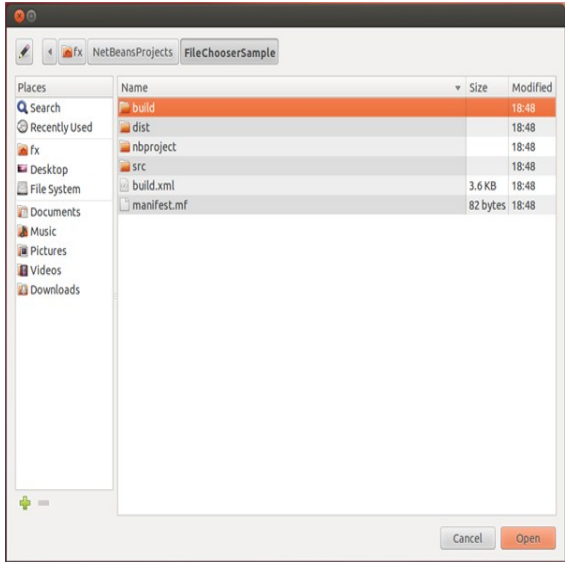
```
val dialog =TextInputDialog(" boîte dialogue")
dialog.title="boîte dialogue"
dialog.headerText="Entrez votre nom:"
dialog.contentText="Nom:"
val resultat = dialog.showAndWait()
// le traitement quand le bouton OK est cliqué
// si "annuler"  alors la boîte se refermera
// on passe en paramètre une lambda qui a un paramètre
resultat.ifPresent({nom ->
      print(nom)
  })
}
```

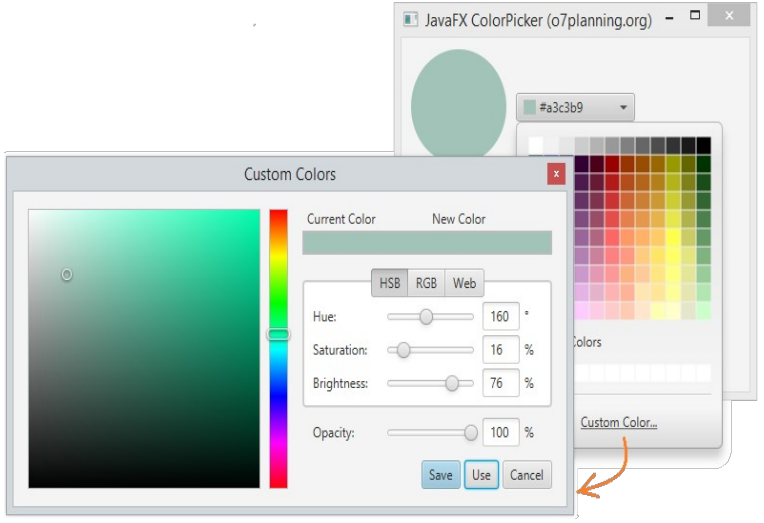


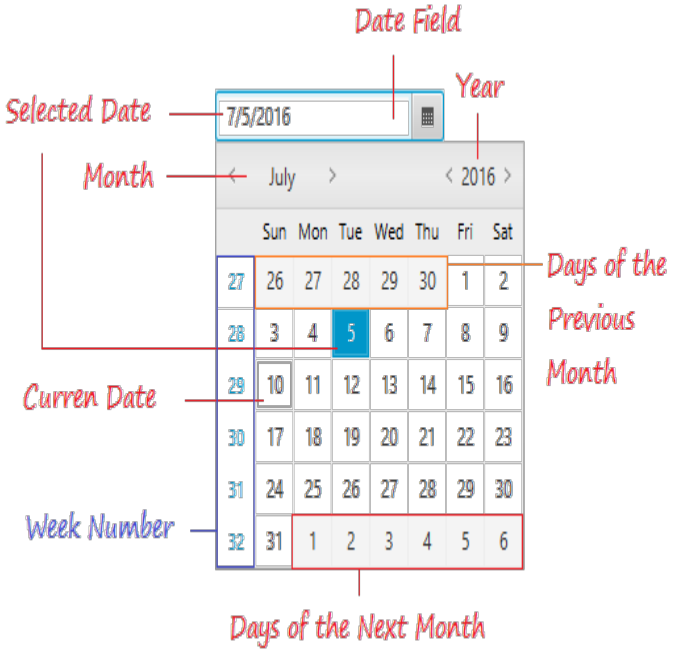


D’autres boîtes de dialogue

FileChooser, DirectoryChooser, DatePicker, ColorPicker







https://devstory.net