

Développement d'application avec IHM

Modèle MVC/binding



Christine Jacquin

Le modèle MVC

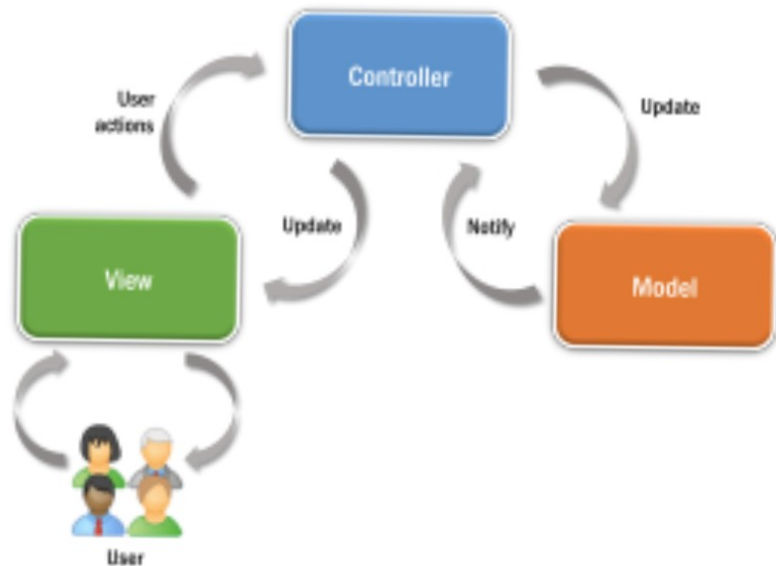
- Une Architecture (ou modèle de conception, design pattern) beaucoup utilisée qui comporte de nombreuses variantes, est qui signifie :

Model - View - Controller

- Dans cette architecture, le code est scindé en entités distinctes (modèles, vues et contrôleurs) qui communiquent entre-elles au moyen de divers mécanismes (invocation de méthodes, génération et réception d'événements, ...).
- Cette architecture a été introduite 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.

Principe de base (1)

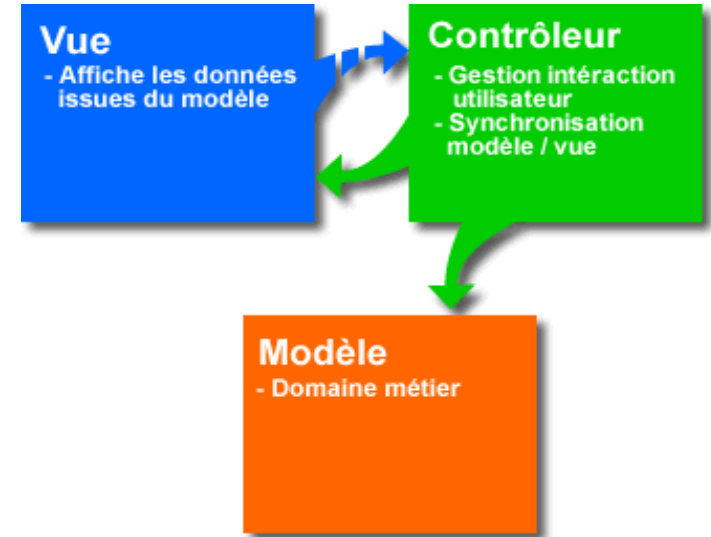
- *Séparer les préoccupations*
- **Modèle** = modèle de données
- **Vue(s)** = description de l'interface
- **Contrôleur** = gestion des événements



Dans cette architecture, le contrôleur voit la vue et le modèle
Par contre, la vue et le modèle ne se connaissent pas

Principe de base (2)

- Le modèle se charge de la gestion des données (accès, transformations, calculs, etc.). Il enregistre (directement ou indirectement) l'état du système et le tient à jour.
 - Les vues représentent l'interface utilisateur (composants, fenêtres, boîtes de dialogue) et ont pour tâche de présenter les informations (visualisation). Les vues participent aussi à la détection de certaines actions de l'utilisateur (clic sur un bouton, appui sur une touche, saisie d'un texte, ...).
- Les contrôleurs sont chargés de réagir aux actions de l'utilisateur (clavier, souris, ...) et à d'autres événements internes (activités en tâches de fond, ...) et externes (réseau, ...).
- Une application peut également comporter du code qui n'est pas directement affecté à l'une de ces trois parties (bibliothèques générales, classes utilitaires, etc.)



[http:// blog.iteratif.fr](http://blog.iteratif.fr)

Architecture d'une application JavaFX MVC

- Le modèle sera fréquemment représenté par une ou plusieurs classes qui peuvent implémenter une interface permettant de s'abstraire des techniques de stockage des données.
- Les vues seront représentées par une ou plusieurs classes. 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 d'expressions lambda.
- La classe principale (celle qui comprend la méthode `start()`) correspond au contrôleur principal.
- D'autres classes utilitaires peuvent venir compléter l'application.

Exemple de la calculatrice

Exemple MVC

Ma super calculatrice

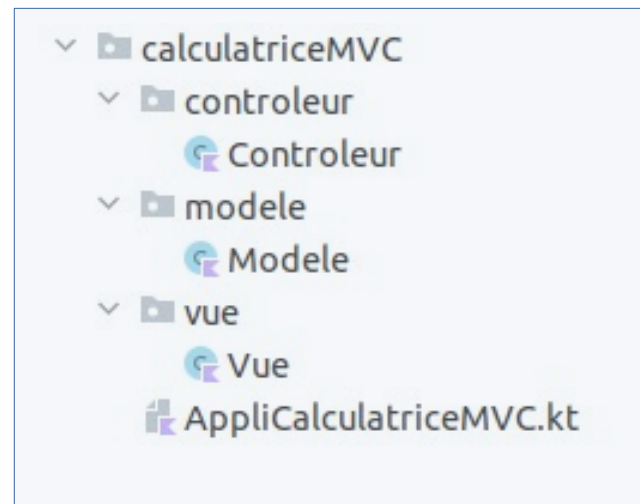
Nombre 1:

Nombre2:

somme

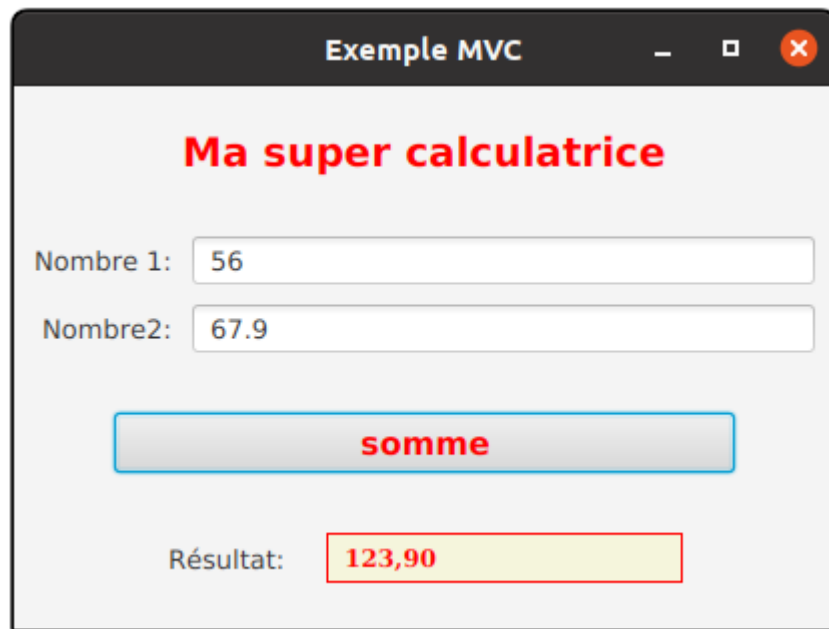
Résultat:

Architecture de l'application



Le code de la vue

```
class Vue(): GridPane(){  
  
    val nombre1= TextField("")  
    val nombre2= TextField("")  
    val resultat= TextField("")  
    val boutonCalcul= Button("somme")  
  
    init{  
  
        val sceneTitle = Label("Ma super calculatrice")  
        this.add(sceneTitle, 0,0,2,1)  
        val nombre1Label = Label("Nombre 1:")  
        this.add(nombre1Label, 0, 2)  
        this.add(nombre1, 1, 2)  
        val nombre2Label = Label("Nombre2:")  
        this.add(nombre2Label, 0, 3)  
        this.add(nombre2, 1, 3)  
        this.add(boutonCalcul,0,4,2,1)  
        val paneResultat= FlowPane()  
        val labelResultat= Label("Résultat:")  
        paneResultat.children.addAll(labelResultat, resultat)  
        this.add(paneResultat,0,5, 2,1)  
  
    }  
}
```



Ajout méthode à la vue

```
class Vue(): GridPane(){
```

```
    val nombre1= TextField("")  
    val nombre2= TextField("")  
    val resultat= TextField()  
    val boutonCalcul= Button("somme")
```

```
    init{
```

```
        val sceneTitle = Label("Ma super calculatrice")  
        this.add(sceneTitle, 0,0,2,1)  
        val nombre1Label = Label("Nombre 1:")  
        this.add(nombre1Label, 0, 2)  
        this.add(nombre1, 1, 2)  
        val nombre2Label = Label("Nombre2:")  
        this.add(nombre2Label, 0, 3)  
        this.add(nombre2, 1, 3)  
        this.add(boutonCalcul,0,4,2,1)  
        val paneResultat= FlowPane()  
        val labelResultat= Label("Résultat:")  
        paneResultat.children.addAll(labelResultat, resultat)  
        this.add(paneResultat,0,5, 2,1)  
    }
```

On va ajouter 2 méthodes pour que le contrôleur puisse : -
- ajouter un gestionnaire d'événement à la vue
- afficher le résultat de la somme dans le *Textfield*

```
fun fixListenerBoutonCalcul(controleur: EventHandler<ActionEvent>) {  
    this.boutonCalcul.onAction=controleur  
}
```

```
fun afficheResultat(retour : Int){  
    this.resultat.text=retour.toString()  
}
```


Le code du modèle

```
class Modele {  
    var nombre1 : Double=0.0  
    var nombre2 : Double=0.0  
  
    fun somme(): Double{  
        return nombre1+nombre2  
    }  
}
```

Le code du contrôleur

```
class Controleur(modele: Modele, vue: Vue): EventHandler<ActionEvent> {
```

```
    private val modele: Modele
```

```
    private val vue: Vue
```

```
    init {  
        this.modele=modele  
        this.vue=vue  
    }
```

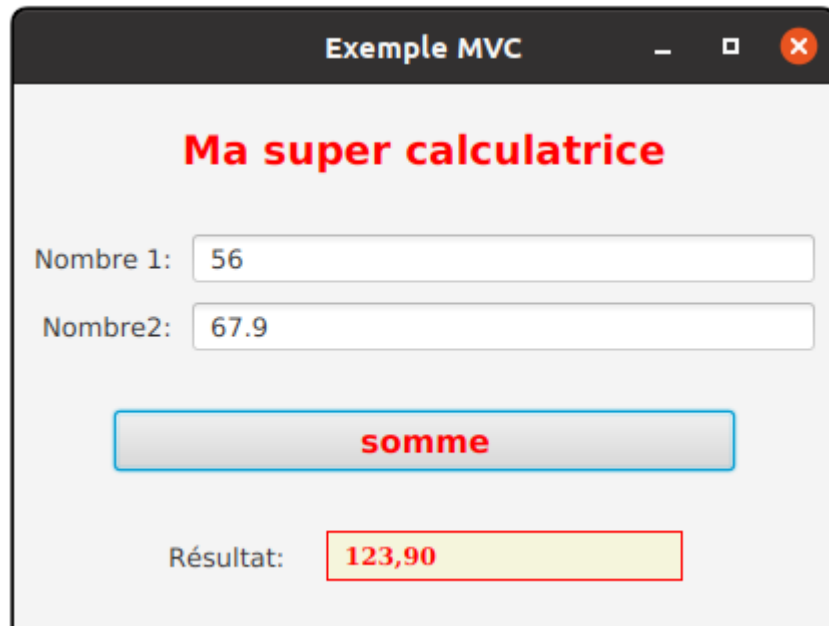
```
    override fun handle(event: ActionEvent) {
```

```
        val nombre1=vue.nombre1.text  
        val nombre2=vue.nombre2.text  
        if ((nombre1!="") && (nombre2!="")){  
            modele.nombre1=nombre1.toDouble()  
            modele.nombre2=nombre2.toDouble()
```

```
        val retour=modele.somme()
```

```
        vue.afficheResultat(retour)
```

```
    }  
}  
}
```



Contrôleur principal

```
class AppliCalculatriceMVC : Application(){
```

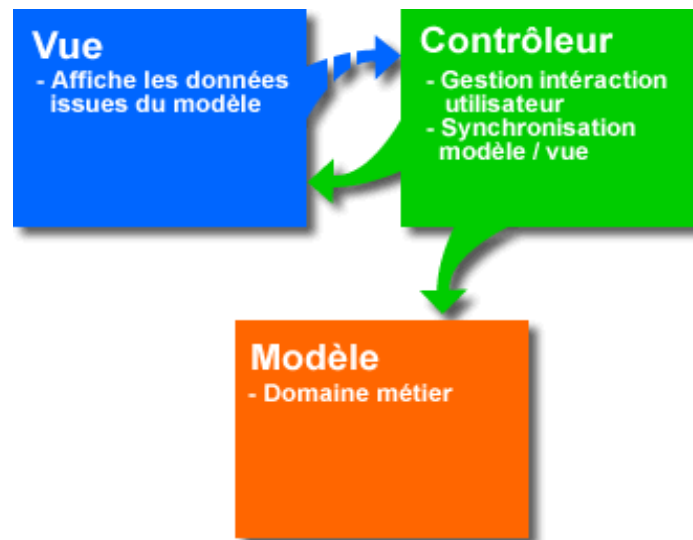
```
override fun start(primaryStage: Stage) {
```

```
    val modele= Modele()  
    val vue= Vue()  
    vue.fixeListenerBoutonCalcul(Contrôleur(modele, vue))  
    val scene = Scene(vue, 400.0, 270.0)  
    primaryStage.title="Exemple MVC"  
    primaryStage.scene=scene  
    primaryStage.show()
```

```
}
```

```
}
```

C'est ici que sont instanciés le modèle et la vue.

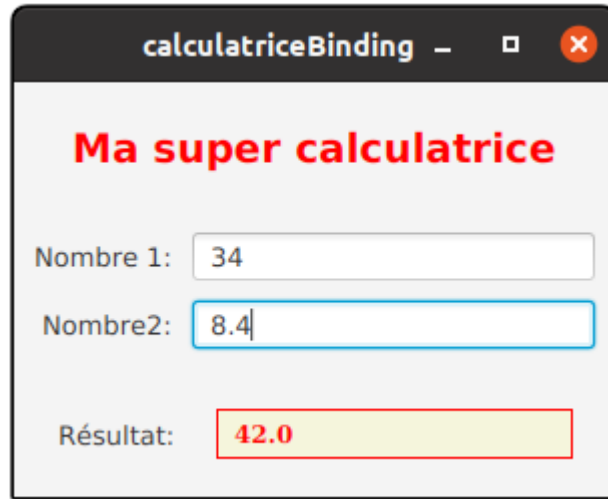


La programmation réactive

- c'est un paradigme de programmation qui vise à conserver une cohérence d'ensemble en propageant les modifications d'une source réactive (modification d'une variable, entrée utilisateur, etc.) aux éléments dépendants de cette source.
- illustration avec le " binding " en JavaFX (les " properties ")

Exemple

Dès que la valeur contenue dans un des deux *TextField* est modifiée, le résultat est modifié



The screenshot shows a Java Swing window titled "calculatriceBinding". Inside the window, the text "Ma super calculatrice" is displayed in red. Below this, there are two input fields: "Nombre 1:" with the value "34" and "Nombre2:" with the value "8.4". The "Nombre2:" field has a blue border, indicating it is the active field. At the bottom, there is a "Résultat:" label followed by a yellow box containing the value "42.0" in red text.

Label	Value
Nombre 1:	34
Nombre2:	8.4
Résultat:	42.0

Les propriétés

En langage Java, la convention dite "JavaBeans", définit qu'une classe possédant une propriété nommée par exemple ***maProp*** comprend une méthode ***getMaProp()*** et ***setMaProp()***.

Les propriétés *JavaFX* possèdent ensuite une troisième méthode ***maPropProperty()*** qui retourne un objet qui implémente l'interface **Property**.

Intérêt des propriétés :

- Elles peuvent déclencher un événement lorsque leur valeur est modifiée et un gestionnaire d'événement peut réagir en conséquence.
- Elles peuvent être liées entre-elles (binding). Cela signifie que la modification d'une valeur d'une propriété entraîne automatiquement la mise à jour d'une autre.

Les propriétés de base

Des propriétés de base existent en javaFX comme :

- `IntegerProperty => SimpleIntegerProperty`
 - `DoubleProperty => SimpleDoubleProperty`
 - `StringProperty => SimpleStringProperty`
 - `ListProperty<E> => SimpleListProperty<E>`
 - `ObjectProperty<T> => SimpleObjectProperty<T>`
-
- La classe abstraite *IntegerProperty* permet d'emballer une valeur de type entier et d'offrir des méthodes pour consulter et modifier la valeur mais également pour "observer" et "lier" les changements.
 - La classe *SimpleIntegerProperty* quant à elle est une classe concrète prédéfinie permettant de créer une telle propriété.

Le mécanisme de binding (1)

Un des avantages des propriétés JavaFX est la possibilité de pouvoir les lier entre-elles. Ce mécanisme, appelé **binding**, permet de mettre à jour automatiquement une propriété en fonction d'une autre.

- Dans les interfaces utilisateurs, on a fréquemment ce type de liens. Par exemple, lorsqu'on déplace le curseur d'un slider, la valeur d'un champ texte changera (ou la luminosité d'une image, la taille d'un graphique, le niveau sonore, ...).
- Il est possible de lier deux propriétés A et B de manière :
 - **Unidirectionnelle**: une modification de A entraînera une modification de B mais pas l'inverse (B ne pourra pas être modifié d'une autre manière).
 - **Bidirectionnelle**: une modification de A entraînera une modification de B et réciproquement (les deux sont modifiables).

Exemple binding unidirectionnel

```
val a: IntegerProperty = SimpleIntegerProperty(15)
```

```
val b: IntegerProperty = SimpleIntegerProperty(10)
```

```
b.bind(a)
```

```
a.set(39)
```

```
print(b.value) => affiche 39
```

```
b.set(50) => le programme plante !!!
```

Exemple binding bidirectionnel

```
val a: IntegerProperty = SimpleIntegerProperty(15)
```

```
val b: IntegerProperty = SimpleIntegerProperty(10)
```

```
b.bindBiDirectional(a)
```

```
a.set(39)
```

```
print(b.value) => affiche 39
```

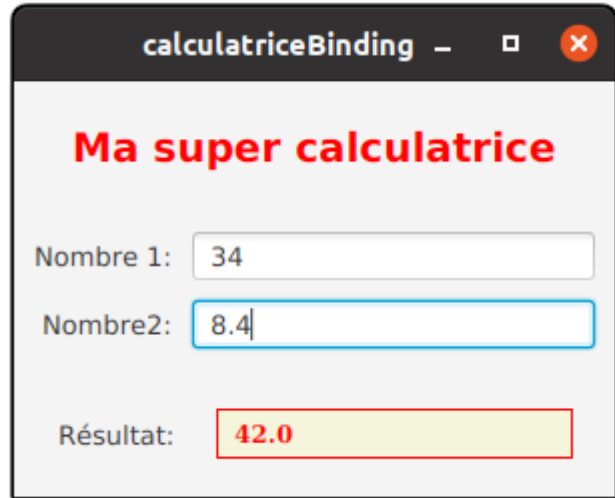
```
b.set(50)
```

```
print(a.value) => affiche 50
```

Le mécanisme de binding (2)

- La méthode **bind(...)** permet de créer un **lien unidirectionnel**. La méthode doit être appelée sur la propriété qui sera "soumise" à l'autre (celle qui est passée en paramètre). Une propriété ne peut être liée qu'à une seule autre si le lien est unidirectionnel.
- La méthode **bindBiDirectional(...)** permet de créer un **lien bidirectionnel**. Toute modification sur une propriété est répercutée sur l'autre et vice versa
- Pour accéder à l'objet empaqueté par une propriété =>
`propriété1.value`
- Parfois, une propriété dépend d'une autre mais avec une relation plus complexe. Il est ainsi possible de créer des propriétés calculées => `propriete1.add(propriete2)`
- Des opérations de conversions sont parfois nécessaires si le type des propriétés à lier n'est pas le même. Par exemple pour lier un champ texte (StringProperty) à un slider dont la valeur est numérique (DoubleProperty).

Exemple de l'application calculatrice avec binding



calculatriceBinding

Ma super calculatrice

Nombre 1: 34

Nombre2: 8.4

Résultat: 42.0

On veut maintenant que toute modification dans les 2 champs texte de saisie soit répercutée dans la zone d'affichage du résultat

Il n'y a plus de bouton somme !!!!!

Mise en place du binding

La vue possède des liaisons (par binding) entre les données d'entrées et la donnée de sortie via le modèle

- L'interface n'a donc plus besoin de bouton pour déclencher le calcul.
- La donnée de sortie est automatiquement mise à jour lorsqu'on change les données d'entrée (durant la saisie dans les champs texte).

Le nouveau modèle

```
class Modele {  
  
    val premierNombre= SimpleDoubleProperty()  
    val secondNombre= SimpleDoubleProperty()  
    val somme=SimpleDoubleProperty()  
  
    fun somme(): Double {  
        return premierNombre.value + secondNombre.value  
    }  
  
}
```

La vue

```
class Vue(): GridPane(){  
  
    val nombre1= TextField("")  
    val nombre2= TextField("")  
    val resultat= TextField()  
  
    init{  
  
        val sceneTitle = Label("Ma super calculatrice")  
        this.add(sceneTitle, 0,0,2,1)  
        val nombre1Label = Label("Nombre 1:")  
        this.add(nombre1Label, 0, 2)  
        this.add(nombre1, 1, 2)  
        val nombre2Label = Label("Nombre2:")  
        this.add(nombre2Label, 0, 3)  
        this.add(nombre2, 1, 3)  
  
        val paneResultat= FlowPane()  
        val labelResultat= Label("Résultat:")  
        paneResultat.children.addAll(labelResultat, resultat)  
        this.add(paneResultat,0,5, 2,1)  
  
    }  
}
```

La vue est identique à la vue précédente (on a simplement supprimé le bouton pour effectuer la somme)

Le nouveau contrôleur

```
class Controller(modele: Modele, vue: Vue){
```

```
    val modele: Modele  
    val vue: Vue
```

```
    init{  
        this.modele=modele  
        this.vue=vue  
    }
```

```
    fun setModeleVue(){
```

```
        // pour convertir les nombres en chaîne de caractère
```

```
        val convertir= NumberStringConverter()
```

```
        // pour que toute modification de l'un, modifie l'autre
```

```
        vue.nombre1.textProperty().bindBidirectional(modele.premierNombre,convertir)
```

```
        vue.nombre2.textProperty().bindBidirectional(modele.secondNombre,convertir)
```

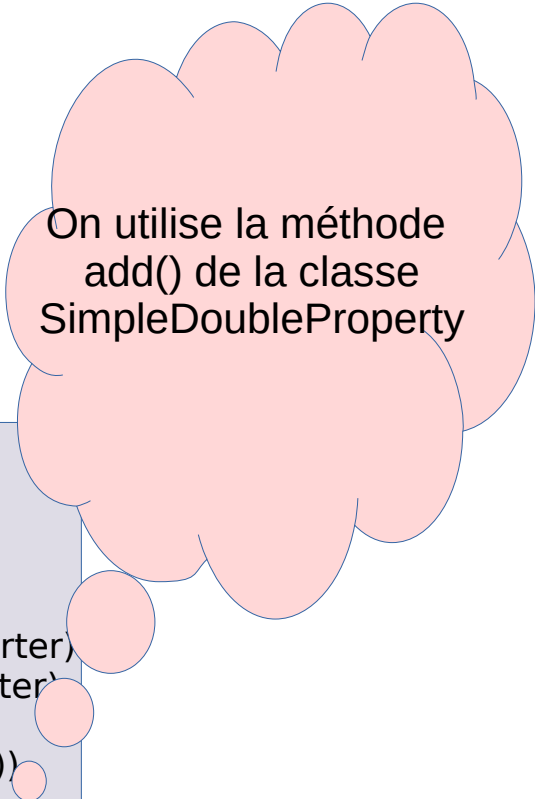
```
        // on lie la propriété somme du modèle au résultat du calcul
```

```
        modele.somme.bind(modele.premierNombre.add(modele.secondNombre))
```

```
        // on bind le textField resultat avec la somme
```

```
        vue.resultat.textProperty().bind(modele.somme.asString())
```

```
    }
```



On utilise la méthode
add() de la classe
SimpleDoubleProperty

Lancement de l'application

```
class MainCalculatrice: Application() {  
  
    override fun start(premierStage: Stage) {  
        premierStage.title = "calculatriceBinding"
```

```
    val vue = Vue()  
    val modele = Modele()
```

Instanciation de la
vue et du modèle

```
    val controleur = Controller(modele, vue)  
    controleur.setModeleVue()
```

Liaison entre le
modèle et la vue

```
    val scene = Scene(vue, 300.0, 300.0)  
    premierStage.scene = scene  
    premierStage.show()
```

```
}
```

Autre manière plus généraliste

Il est possible en *javaFX* de réaliser des liaisons (binding) de plusieurs manières différentes. La plus simple vous a été montré (binding de haut-niveau : **fluent API**)

Si on doit développer des calculs plus complexes, il est possible de mettre en place des liaisons de bas-niveau (**low-level bindings**) entre les propriétés des divers éléments

- Pour créer des liaisons de bas-niveau, il faut redéfinir les méthodes *computeValue()* de liaisons existantes (on crée des sous-classes de *IntegerBinding*, *DoubleBinding*, *StringBinding*, ...)

Pour plus d'information:

<https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>

Les classes observables

L'interface **Observable** est l'interface mère de nombreuses interfaces qui sont aussi des **Collection**.

- `ObservableList<E>`
- `ObservableArray<T>`
- `ObservableMap<K,P>`
- `ObservableSet<E> ...`

- La spécificité des instances des classes qui implémentent ces interfaces est que lorsqu'une modification, un ajout ou une suppression d'élément a lieu en leur sein (modèle), elles notifient le composant graphique qui les "observe" (s'il a cette possibilité) et celui-ci répercute les modifications.
- La classe `FXCollection` permet d'obtenir des instances de ces classes

ObservableList et ListView (1)

```
val countries: ObservableList<String>
val capitals: ObservableList<String>
val countriesListView: ListView<String>
val capitalsListView: ListView<String>
val leftButton: Button
val rightButton: Button
```

```
init {
```

```
countries = FXCollections.observableArrayList(
    "Australia", "Vienna", "Canberra", "Austria", "Belgium", "Santiago", "Chile", "Brussels", "San Jose", "Finland", "India")
```

```
capitals = FXCollections.observableArrayList(
    "Costa Rica", "New Delhi", "Washington DC", "USA", "UK", "London", "Helsinki", "Taiwan", "Taipei", "Sweden", "Stockholm")
```

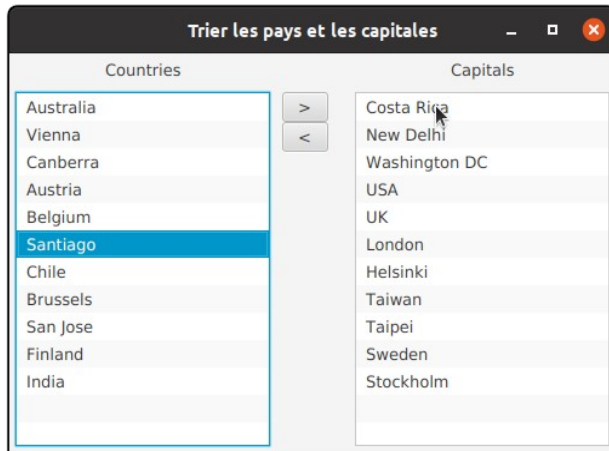
```
countriesListView = ListView<String>(countries)
```

```
capitalsListView = ListView<String>(capitals)
```

```
leftButton = Button("< ")
leftButton.onAction=ButtonEcouleur(this)
```

```
rightButton = Button(" > ");
rightButton.onAction=ButtonEcouleur(this)
```

```
}
```



ObservableList et ListView (2)

```
class ButtonEcouleur(appli: Main) : EventHandler<ActionEvent> {
```

```
val appli: Main = appli
```

```
override fun handle(event: ActionEvent) {
```

```
    if (event.source.equals(appli.leftButton)) {
```

```
        val str = appli.capitalsListView.selectionModel.selectedItem
```

```
        if (str != null) {
```

```
            appli.capitals.remove(str)
```

```
            appli.countries.add(str)
```

```
        }
```

```
    } else if (event.source.equals(appli.rightButton)) {
```

```
        val str = appli.countriesListView.selectionModel.selectedItem
```

```
        if (str != null) {
```

```
            appli.countriesListView.selectionModel.clearSelection()
```

```
            appli.countries.remove(str)
```

```
            appli.capitals.add(str)
```

```
        }
```

```
    }
```

```
}
```



Mise à jour des listes de capitale et de pays

Mise à jour des listes de capitale et de pays