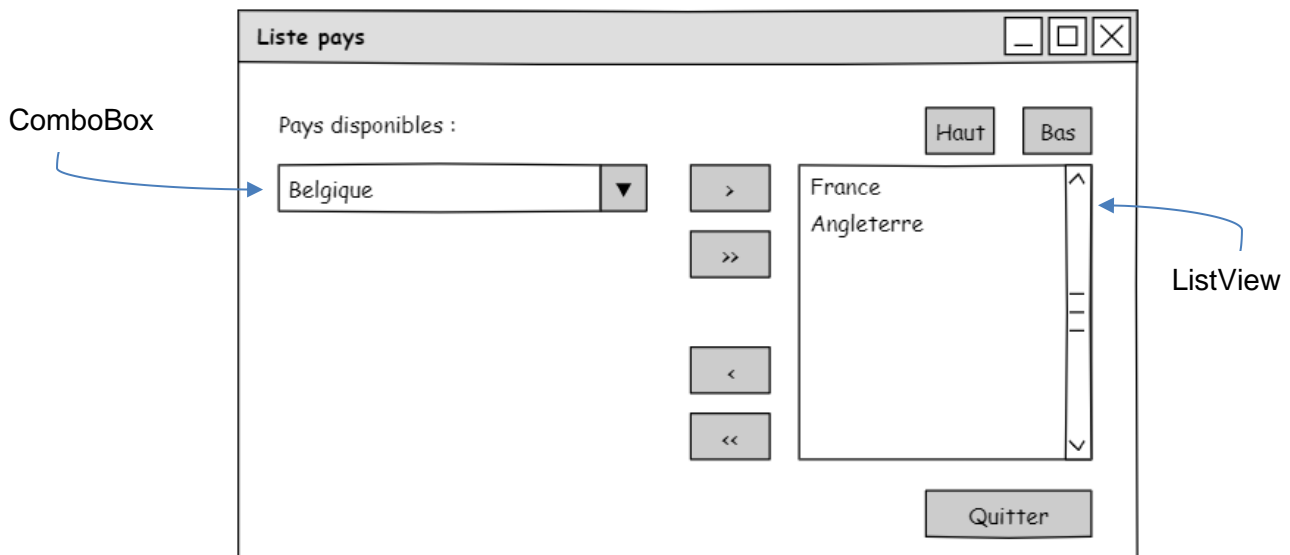






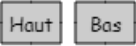
1. FENETRE A DEVELOPPER

L'objectif de ce programme est de gérer le contenu d'une liste (« **ListView** ») et d'une liste déroulante (« **ComboBox** ») via plusieurs boutons dont le fonctionnement sera détaillé plus loin.

Ci-dessous un wireframe du résultat attendu :



1.1 OPERATIONS ATTENDUES

-  permet d'ajouter l'élément sélectionné dans la combobox dans la liste. Il faudra veiller à supprimer l'élément de la combobox. Ce bouton doit être désactivé si il n'y a plus d'éléments dans la combobox ;
-  permet d'ajouter tous les éléments de la combobox dans la liste. Ce bouton doit être désactivé si la combobox est vide ;
-  permet de supprimer l'élément sélectionné de la liste et de le remettre dans la combobox (doit être désactivé sur la liste est vide) ;
-  permet de supprimer tous les éléments de la liste et de les remettre dans la liste (doit être désactivé sur la liste est vide) ;
-  permettent respectivement de monter un élément sélectionné de la liste d'un cran et de descendre un élément d'un cran.

La « **ListView** » devra être équipée d'une barre de défilement (ou « **scrollbar** ») au cas où la liste des pays est de grande taille.

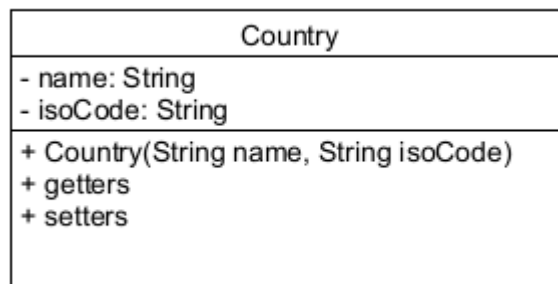
1.2 MANIPULATION D'OBJETS

L'objectif est ici de gérer une liste de pays. Vous pourriez manipuler des « String » et les ajouter aux différents composants.

Mais il est également possible de permettre aux « ComboBox » ou « ListView » de se baser sur d'autres types d'objets.

Pour la suite vous allez créer une classe « **Country** » représentant un pays.

Voici le diagramme UML de la classe à implémenter :



Le code ISO est un identifiant à 3 chiffres défini par le [standard ISO](#) (« International Organization for Standardization »).

1.2.1 Ajout d'objets à une « ComboBox » ou une « ListView »

Il est possible d'ajouter et d'afficher des objets dans les composants graphiques de type « **ComboBox** » et « **ListView** ».

Pour se faire il vous faut créer une liste particulière, une « **ObservableList** ».

« **ObservableList<T>** » pouvant être instanciée à l'aide de la méthode statique de la classe « **FXCollections** » appelée « **observableArrayList** ».

Extrait de la [documentation](#) :

```
public static <E> ObservableList<E> observableArrayList()
```

Creates a new empty observable list that is backed by an arraylist.

Returns:

a newly created ObservableList

See Also:

[observableList\(java.util.List\)](#)

« **ObservableList<E>** » est une **classe générique** pouvant être instanciée avec n'importe quel objet (il suffit de préciser le nom de la classe entre les chevrons).

Par exemple, il vous est possible de créer une liste observable et de l'associer à une « ComboBox » de la façon suivante :

```
ObservableList<Country> comboBoxList = FXCollections.observableArrayList();
comboBoxList.add(new Country("Ukraine", "UKR"));
comboBoxList.add(new Country("Argentine", "ARG"));
ComboBox<Country> comboBox = new ComboBox<Country>(comboBoxList);
```

Vous pourrez utiliser la liste tout comme une « **ArrayList** », en utilisant les fonctions suivantes (et bien d'autres) :

- [add\(E e\)](#) : permet d'ajouter un élément
- [get\(int index\)](#) : permet de récupérer un élément à l'index indiqué
- [remove\(int index\)](#) : permet de supprimer un élément

Ces méthodes sont héritées de plusieurs interfaces : [Collection<E>](#) et [List<E>](#)

Par la suite, il sera possible « d'écouter » les changements en utilisant la méthode « [addListener\(\)](#) ».

Extrait de documentation :

```
void addListener(ListChangeListener<? super E> listener)
```

Add a listener to this observable list.

Parameters:

`listener` - the listener for listening to the list changes

Un « listener » est, basiquement, similaire à un « event handler » et permettra de gérer les modifications sur la liste.

Voici un exemple de code pouvant être utilisé pour associer un « listener » à une liste observable (code à ajouter dans une fonction).

```
comboBoxList.addListener(new ListChangeListener<Country>() {
    @Override
    public void onChanged(Change<? extends Country> country) {
        System.out.print("Il s'est passé un truc, là.");
    }
});
```

1.2.2 Chargement d'un grand ensemble de pays

Vous pourrez utiliser la fonction suivante pour instancier les pays (essayer d'analyser le code afin de comprendre comment ça fonctionne) :

```
public static ArrayList<Country> getCountriesList() {
    ArrayList<Country> countries = new ArrayList<Country>();

    String[] countryCodes = Locale.getISOCountries();

    for (String countryCode : countryCodes) {
        Locale obj = new Locale("", countryCode);
        countries.add(new Country(obj.getDisplayCountry(), obj.getISO3Country()));
    }

    return countries;
}
```

1.3 IMAGES SUR LES BOUTONS DE MANIPULATION DE LA LISTE

Afin de rendre l'interface utilisateur plus attractive et les fonctionnalités plus explicites il vous est possible d'ajouter des images aux boutons (<https://devstory.net/11091/javafx-button>).

Ajouter les images suivantes pour les boutons « Haut » et « Bas » :

- <https://www.iconfinder.com/icons/211623/download/png/512>
- <https://www.iconfinder.com/icons/211614/download/png/512>

Pour intégrer des fichiers utilisables au sein du code Java vous pouvez utiliser le système de « resource » de Java (<https://www.javatpoint.com/java-class-getresource-method>).

Si vous votre projet a été créé avec l'archétype « **javafx-archetype-fxml** » il vous est possible d'ajouter les images dans le dossier nommé « **ressources** ».

Voici un exemple de code pour charger un fichier d'un dossier « resource » et de créer un objet de la classe « **ImageView** » utilisable avec un bouton.

```
FileInputStream imageFileInputStream = new
FileInputStream(getClass().getResource("media/up_arrow.png").getPath());
Image image = new Image(imageFileInputStream);
ImageView imageView = new ImageView(image);
Button button = new Button("Bouton", imageView);
```

1.4 ARCHITECTURE LOGICIELLE

Afin de garder un code lisible et maintenable, il est fortement conseillé de séparer en plusieurs classes les éléments graphiques (et donc plusieurs fichiers).

Pour se faire il faut se poser la question suivante : quelle section de ma fenêtre pourrait être utilisée de façon autonome ?

Proposition de classes pour l'organisation du code :

- « **App** » : va correspondre à la fenêtre principale et contenir tous les composants de base
- « **UpDownListView<T>** » : va correspondre à la « **ListView** » avec ses boutons de positionnement des objets ;
- « **Country** » : objet métier pays ;

Voici une ébauche de la **classe générique** « **UpDownListView<T>** » à faire évoluer :

```
/**
 * Composant graphique permettant de gérer une ListView d'objets
 * hérite de VBox pour obtenir l'interface souhaitée.
 */
public class UpDownListView<T> extends VBox {

    private Button upBtn;
    private Button downBtn;
    private ListView<T> listView;

    /**
     * Cette liste sera instanciée par un code extérieur à la classe
     * Dans App.java, par exemple :)
     */
    private ObservableList<T> objectsList;

    public UpDownListView() {
        // TODO Charger les images des boutons
        // -> code fourni dans l'énoncé du TP

        // Création des composants graphiques utilisés
        this.upBtn = new Button("Up");
        this.downBtn = new Button("Down");
        this.listView = new ListView<T>();

        // positionnement des boutons côte-à-côte, utilisation d'un HBox
        HBox buttonBox = new HBox();
        buttonBox.setSpacing(5);
        buttonBox.getChildren().add(this.upBtn);
        buttonBox.getChildren().add(this.downBtn);
        buttonBox.setAlignment(Pos.CENTER_RIGHT);
    }
}
```

```

        this.getChildren().add(buttonBox);
        this.getChildren().add(this.listView);
    }

    /**
     * Fonction à appeler pour associer une "ObservableList" à ce composant
    graphique (sinon ça marche pas).
     * @param list La liste observable concernée.
     */
    public void setObjectsList(ObservableList<T> list) {
        this.objectsList = list;
        listView.setItems(list);
    }

    public ObservableList<T> getObjectsList() {
        return objectsList;
    }
}

```

CREDITS

ŒUVRE COLLECTIVE DE L'AFPA

Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)

Michel Coulard – Formateur Evry

Chantal Perrachon – IF Neuilly sur Marne

Date de mise à jour : 19/02/2023

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconque. »