TP - Une interface graphique simple en JavaFX

Installer JavaFX

Dans un premier temps, il vous faut pouvoir utiliser JavaFX, car cette librairie est désormais dissociée du JDK depuis Java 11. En principe, votre IDE intègre déjà javaFX. Pour le vérifier, soit vous avez la possibilité de créer directement un nouveau projet JavaFX, soit, depuis votre projet existant, l'instruction

```
import javafx.application.Application;
```

compile correctement. Si ce n'est pas le cas, il vous faut dans un premier temps télécharger JavaFX depuis le projet officiel : https://openjfx.io/
Il est obligatoire de télécharger la même version que votre version de Java (dans la plupart des cas, ce sera la 11). Ensuite, utilisez la documentation de votre IDE afin de pouvoir utiliser JavaFX avec celui-ci. Des tutoriels sont aussi disponibles pour les principaux IDE :

- Eclipse : https://www.javatpoint.com/javafx-with-eclipse
- Netbeans : https://netbeans.org/kb/72/java/javafx-setup.html
- IntelliJ IDEA : https://www.jetbrains.com/help/idea/javafx.html

Débuter avec JavaFX

On suppose ici que le TP précédent est complet et fonctionne. Soit, à l'intérieur de votre projet existant, vous pouvez utiliser JavaFX (l'instruction import plus haut doit compiler), soit il vous faut créer un nouveau projet JavaFX, et y copier les sources de votre TP précédent.

La classe principale qui va lancer l'affichage (celle contenant un main) doit hériter de la classe **`Application`**. Appelons la **`MainGui`**. Hériter de cette classe vous oblige à implémenter la méthode

```
public void start(Stage stage) throws Exception
```

Dans laquelle vous allez initialiser les éléments de votre application, en particulier :

- Elle prend en paramètre un objet de type **`Stage`**, représentant la fenêtre de votre application. Vous pouvez ainsi modifier la taille de la fenêtre grâce aux méthodes **`setWidth`** et **`setHeight`** de cet objet, ainsi que le titre, avec la méthode **`setTitle`**.

- Nos éléments seront insérés dans notre application par le biais d'un objet de type **`Group`**, qui peut être vu comme un arbre dont les fils sont soit des formes (sous-classes de la classe **`Shape`**, par exemple : **`Rectangle`**, **`Circle`**, **`Text`**...etc), soit des autres groupes (objets de type Group). On peut faire l'analogie avec un système de fichiers : **`Group`** peut être vu comme un dossier, les **`Shape`** comme des fichiers : un dossier peut contenir des fichiers ou d'autres dossiers.

- Lors de l'initialisation de votre application (méthode start), vous devez alors créer :

  - un groupe racine (appelé root par la suite)

```
Group root = new Group();
```

- un objet de type Scene, initialisé à l'aide de root. On le construit ainsi :

```
Scene scene = new Scene(root);
```

Vous pouvez par exemple spécifier la couleur de l'arrière plan de votre application grâce à la méthode **`setFill`** de la classe **`Scene`**.

- Vous pouvez ensuite créer les différentes parties de votre application, et les ajouter de la manière suivante :

```
root.getChildren().add(s);
```

où s est une forme (sous-classe de **`Shape`**), ou un autre groupe (en fait, n'importe quoi héritant de la classe **`Node`**, classe mère de **`Shape`**). Essayez par exemple de créer un texte et de l'ajouter à la racine :

```
Text text = new Text(10, 30, "Hello world");
root.getChildren().add(text);
```

Dans le constructeur, 10 et 30 désignent les coordonnées x et y de notre objet texte. Une fois nos éléments rajoutés à **`root`**, on ajoute la scène à l'attribut **`stage`**, et on affiche cette dernière.

```
stage.setScene(scene);
stage.show();
```

Enfin, la méthode **`main`** sera la suivante :

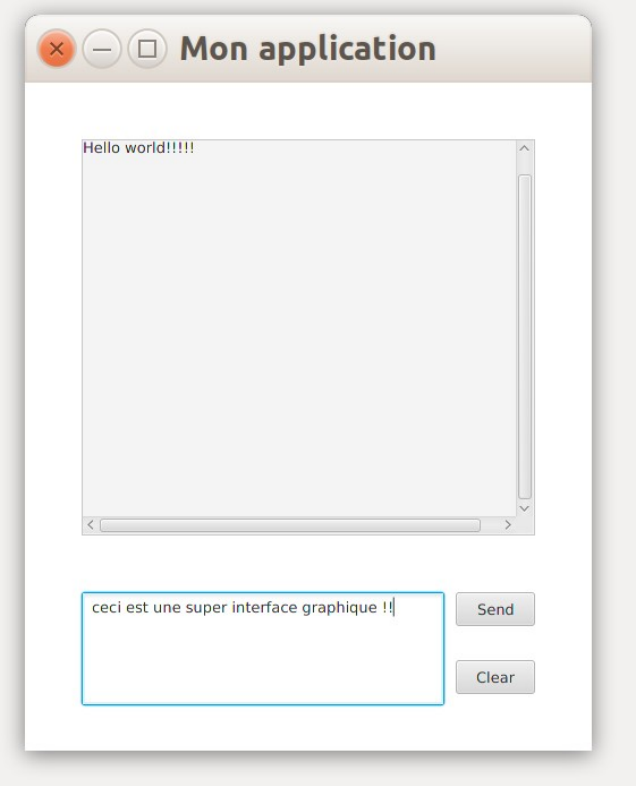
```
public static void main(String[] args) {

        Application.launch(MainGUI.class, args);

 }
```

Notre application :

On se propose de réaliser une interface graphique simple pour la partie client du TP précédent :



Note 1 : des outils graphiques (de type WYSIWYG : What You See Is What You Get) permettant de construire facilement des interfaces graphiques existent, comme par exemple SceneBuilder développé par Gluon. Cependant, afin de comprendre comment les choses fonctionnent, nous allons ici construire notre application avec des lignes de code java seulement. Libre à vous ensuite d'utiliser SceneBuiler (il peut par exemple s'utiliser à partir d'Éclipse ou Netbeans).

Note 2 : nous ne mentionnerons jamais les librairies à inclure afin que votre code compile. Cependant, tout bon IDE vous proposera d'inclure automatiquement certaines d'entre elles. Parfois, des classes ayant le même nom existent dans plusieurs librairie différentes. Dans ce cas, veillez toujours à inclure les librairies commençant par **javafx.**

Note 3 : dans cette partie, vous ne devez créer que 2 classes seulement (en plus de vos classes précédentes qui font fonctionner l'application réseau) : une classe contenant votre main (appelée **MainGui**), et une classe **ClientPanel** (voir plus bas). Autrement dit : toutes les autres classes mentionnées (`Label`, `Button`, `TextFlow`...) font partie de la librairie javafx, vous ne devez pas les recréer !

Comme nous pouvons le voir, l'application est composée de quatre éléments :

- une zone de texte permettant de saisir du texte
- une zone de texte affichant les messages reçus
- deux boutons : un permettant d'envoyer le texte (pour l'instant, il n'enverra le texte que sur la zone des messages reçus), et un permettant d'effacer la zone de saisie

Dans votre package client, créez une classe **ClientPanel** qui va construire tous ces objets. Cette classe héritera de la classe javafx **Parent** (elle même héritant de **Node**, mentionnée précédemment). Puis, dans votre classe qui contient le main (celle qui hérite de **Application**), et plus précisément dans la méthode **start(Stage stage)**, instanciez un objet de type **ClientPanel**, et ajoutez le à votre groupe **root**. Enfin, modifiez la taille de votre fenêtre afin qu'elle fasse 600px de large pour 500px de hauteur. On obtient ainsi la méthode **start** suivante :

```
public void start(Stage stage) throws Exception {

        ClientPanel clientPanel = new ClientPanel();

        Group root = new Group();

        root.getChildren().add(clientPanel);

        Scene scene = new Scene(root, 600, 500);

        stage.setTitle("Mon application");

        stage.setScene(scene);

        stage.show();

}
```

En principe, une fenêtre vide s'affiche. C'est normal, la classe **ClientPanel** est vide.

Dans la classe **ClientPanel**, nous allons ajouter les attributs de classe suivants :

1) Zone de texte permettant de saisir du texte :
- un objet de type **TextArea** appelé *textToSend*

2) Zone de texte affichant les messages reçus :
- un objet de type **ScrollPane** appelé *scrollReceivedText*
- un objet de type **TextFlow** appelé *receivedText*

3) Bouton permettant d'envoyer du texte
- un objet de type **Button** appelé *sendBtn*

4) Bouton permettant d'effacer la zone de saisie
- un objet de type **Button** appelé *clearBtn*

Dans le constructeur de **ClientPanel**, initialisez ces 5 attributs (en utilisant le constructeur vide de chaque classe correspondante). Puis, à la fin de votre constructeur, ajoutez ces éléments à votre objet, **sauf l'élément *receivedText*** (car celui-ci sera encapsulé dans *scrollReceivedText*):

```
this.getChildren().add(scrollReceivedText);
this.getChildren().add(textToSend);
this.getChildren().add(clearBtn);
this.getChildren().add(sendBtn);
```

Si vous lancez votre application, elle ne doit probablement pas ressembler à grand-chose. C'est normal. Toujours dans le constructeur, entre les initialisations des attributs et leurs ajouts à l'objet, il faut spécifier leurs paramètres : abscisse, ordonnée, taille, contenu.

Pour chacun d'entre eux, fixez une abscisse et une ordonnée à l'aide des méthodes **setLayoutX** et **setLayoutY**. Donnez leur également une longueur et une largeur à l'aide des méthodes **setPrefHeight** et **setPrefWidth**

Pour les valeurs d'abscisse, d'ordonnée, de longueur et largeur, prenez en compte les remarques suivantes :

- l'origine (0, 0) est le coin en haut à gauche de votre application
- une abscisse positive vous fera avancer vers la droite à partir de l'origine
- une ordonnée positive vous fera avancer vers le bas à partir de l'origine

Par exemple, pour l'objet *scrollReceivedText*, on aurait :

```
scrollReceivedText = new ScrollPane();
scrollReceivedText.setLayoutX(50);
scrollReceivedText.setLayoutY(50);
scrollReceivedText.setPrefWidth(400);
scrollReceivedText.setPrefHeight(350);
```

Une exception concerne l'objet *receivedText*, qui sera encapsulé par l'objet *scrollReceivedText*. Il faut lui spécifier une largeur (la même que `scrollReceivedText`), mais il aura une abscisse et une ordonnée nulles. En revanche on indiquera au *scrollReceivedText* qu'il le contient. On va également lier le défilement (scroll) de l'objet *scrollReceivedText* à la hauteur de l'objet *receivedText*, une astuce afin de pouvoir toujours lire les derniers messages reçus (pour automatiquement "scroller vers le bas"). Ceci se fait à l'aide des deux instructions suivantes :

```
scrollReceivedText.setContent(receivedText);
scrollReceivedText.vvalueProperty().bind(receivedText.heightProperty());
```

Enfin, les objets de type **Button** et **TextFlow** (c'est à dire dans notre cas les objets *receivedText, sendBtn* et *clearBtn*) nécessitent d'indiquer explicitement qu'ils seront visibles, à l'aide de la méthode **setVisible** qui prend en paramètre un booléen.

Si vous lancez votre application, et certainement après quelques ajustement de tailles et de positionnement, vous devrez avoir un rendu similaire à celui demandé.

La dernière étape est de rendre les boutons utilisables. Pour cela, nous allons utiliser la notion d'événements en JavaFX (notion qui existe dans toutes les librairies permettant l'utilisation d'interfaces graphiques). Presque tous les éléments d'une interface graphique peuvent réagir à toute sorte d'événements : survol par le curseur, clic de souris, bouton pressé, touche de clavier pressée...etc. Pour chacun de ces événements affectant un objet, il faut lui spécifier un objet qui va écouter cet événement. On réalise ceci avec les méthodes de la famille **setOnQuelquechose**. Par exemple, pour nos deux boutons, nous allons spécifier un écouteur lors du clic grâce à la méthode **setOnAction**. Cette méthode attend un objet de type **EventHandler<ActionEvent>**. EventHandler est en fait une interface ne contenant qu'une seule méthode **handle(ActionEven event)** qui définira le code à exécuter lors de l'action (le paramètre **event** contient des informations sur l'événement : coordonnées du curseur pour des événements de souris, touche pressée pour des événements de clavier...etc).

Afin de simplifier le code (il nous faudrait en principe créer une classe implémentant l'interface **EventHandler**, et instancier un tel objet pour le passer en paramètre de la méthode **setOnAction**, et ce pour chaque écouteur), nous allons créer un objet de ce type en définissant une classe interne anonyme de la manière suivante :

```
monBouton.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                //action à effectuer
```

```
            }
});
```

Evidemment, il faut remplacer **monBouton** par **clearBtn** et **sendBtn**. Pour le bouton **clearBtn**, il suffit de remettre à zéro le texte de l'objet **textToSend** (grâce à la méthode **setText**). Quant au bouton **sentBtn**, il nous faut faire deux choses : rajouter le texte dans la zone **receivedText**, mais aussi envoyer le message par le réseau. Pour la première tâche, nous allons créer une méthode qui affiche dans **receivedText** un nouveau message (on crée une méthode à part car celle-ci sera aussi utilisée lors de la réception d'un message par le réseau).

On crée donc une méthode **printNewMessage** dont le code est donné ci-après :

```
public void printNewMessage(Message mess) {

        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                Label text = new Label("\n" + mess.toString());
                text.setPrefWidth(receivedText.getPrefWidth() - 20);
                text.setAlignment(Pos.CENTER_LEFT);
                receivedText.getChildren().add(text);
            }
        });
    }
```

Quelques explications :

- on encapsule tout le code dans un appel à **Platform.runLater**. À l'intérieur, vous reconnaissez probablement le code pour lancer un nouveau thread (en définissant une classe anonyme qui implémente l'interface **Runnable**). Ce code va en effet changer l'interface graphique gérée par JavaFX, et sera appelé, une fois votre application terminée, soit du thread de l'interface graphique (lorsque vous cliquez sur le bouton "send"), soit de votre thread qui recevra des nouveaux messages (la classe **ClientReceive**). Cependant, un thread autre que ceux gérés par JavaFX ne peut pas modifier l'interface graphique (c'est une contrainte technique de JavaFX). Pour résoudre ce problème, on lance ainsi un nouveau thread à part, et on indique à JavaFX de traiter ce thread lorsqu'il le décidera

- on crée un nouvel objet graphique de type **Label**, et on l'ajoute à notre objet **receivedText**

Si on revient maintenant au code de l'écouteur de notre bouton **sendButton** (la méthode **handle** vue plus haut), on doit faire quatre choses :

1) créer un objet **Message** : le **sender** sera la chaîne "Moi", par exemple, et le contenu du message sera le contenu textuel de notre objet **textToSend**, accessible via **textToSend.getText()**
2) appeler notre méthode **printNewMessage** avec ce message
3) remettre à zéro le contenu de l'objet **textToSend**, grâce à la méthode **setText**
4) envoyer le message en réseau

Pour la tâche 4), le but est d'invoquer la méthode **sendMessage** de notre classe **Client**, qui fait déjà tout le travail. Cependant, nous n'avons pas accès à l'instance de Client (en fait, notre nouveau **Main** ne le crée même pas). Il faut ainsi rajouter une relation entre notre classe **Client** et notre classe **ClientPanel**. Cette relation doit se faire dans les deux sens, car :

- la vue (**ClientPanel**) a besoin d'accéder au **Client** pour envoyer des messages (c'est la tâche 4 ci-dessus)

- le **Client** a aussi besoin d'accéder à la vue : quand le thread **ClientReceive**

reçoit un message, il appelle en principe la méthode **`messageReceived`** de la classe **`Client`**. Jusqu'à présent, cette méthode se contentait d'afficher le message reçu avec un **`System.out.println`**. Désormais, on veut invoquer notre nouvelle méthode **`printNewMessage`** de notre **`ClientPanel`**. On va ainsi créer une relation bidirectionnelle entre **`ClientPanel`** (la vue) et **`Client`** (le controleur).

Ce qu'il faut faire :

- dans la classe **`ClientPanel`**, créer un attribut **client** de type **`Client`**
- dans la classe **`Client`**, créer un attribut **view** de type **`ClientPanel`**
- créer dans ces deux classes un setter permettant d'initialiser ces attributs
- dans votre main (méthode **`start()`** de la classe **`MainView`**) : créez un objet de type **`Client`**, et, en utilisant vos setters :
  - affectez ce **`Client`** à l'objet **`ClientPanel`**
  - affectez le **`ClientPanel`** à l'objet **`Client`**

Remarque : la création de l'objet **`Client`** nécessite l'adresse IP et le port du serveur, qui étaient en principe récupérés depuis les arguments du programme. Pour ce nouveau main, deux solutions :

- une solution rapide temporaire : stockez l'adresse IP et le port "en dur" dans votre code avec deux variables
- pour récupérer les arguments du programme "comme avant", une petite manipulation est nécessaire lorsqu'on utilise JavaFX, puisque l'application est lancée avec l'instruction

```
Application.launch(MainView.class, args);
```

Les arguments sont bien passés au thread **`MainView`**. Cependant, pour les récupérer, dans votre méthode **`start`**, vous utiliserez la méthode **`getParameters`** de la classe **`Application`** :

```
String address = this.getParameters().getRaw().get(0);
int port = Integer.parseInt(this.getParameters().getRaw().get(1));
```

Afin de terminer votre application, vous pouvez maintenant faire les choses suivantes :

- dans le constructeur de **`Client`**, commentez la partie du code qui lance le thread **`ClientSend`**, puisque c'est l'interface graphique qui jouera ce rôle désormais.

- dans notre écouteur du clic de bouton d'envoi (méthode **`handle`** vue plus haut), rajouter un appel à **`sendMessage`** de l'attribut **`client`**, afin d'envoyer le message par le réseau
- dans notre méthode **`messageReceived`** de **`Client`**, invoquer la méthode **`printNewMessage`** de l'attribut view