Interfaces graphiques en Java

Pour des raison historiques, plusieurs bibliothèques sont utilisables (y compris dans le JDK) :

- AWT : existe depuis les premières versions de Java, se repose sur les composants graphiques "natifs" du système d'exploitation (rapide, mais apparence différente entre Windows, macOS, Linux, etc...)
- Swing : bibliothèque "officielle" de Java. Dépend peu des composants du système (donc apparence différente entre plateformes).
- SWT (et surcouche JFace) : bibliothèque du projet Eclipse. Se repose principalement sur les composants natifs (comme AWT) mais implémente tout ce que le système ne fournit pas.
- JavaFX : alternative plus moderne, similaire à Swing dans les principes. [1]

Dans ce cours : exemple de JavaFX, mais principes similaires pour les autres.

1. Intégrée un temps au JDK comme potentiel successeur de Swing (Java 8), puis finalement confiée au projet OpenJFX (Java 11).

Compléments en POO
Aldric Degorre
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Concurrence
Interfaces graphiques
Interfaces graphiques
Principes
JavaFX
Stratégies
Gestion des erreurs et exceptions

Interfaces graphiques en Java

Avertissement

Attention, c'est un sujet très vaste, même en se limitant à JavaFX.

Ce cours ne fera qu'effleurer certains des sujets essentiels et donner quelques pointeurs pour mener à bien le projet.

Il conviendra d'avoir une démarche active pour combler d'éventuels besoins non couverts par le cours (consultez les pages de documentation de Java !!!).

Compléments en POO
Aldric Degorre
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Concurrence
Interfaces graphiques
Interfaces graphiques
Principes
JavaFX
Stratégies
Gestion des erreurs et exceptions

Principes généraux

- **Interface graphique :** hiérarchie de **composants** graphiques se contenant les uns les autres (ex : un bouton dans un panneau dans une fenêtre...)
- Quelques composants standard (fournis par l'API), mais en général on aime bien les personnaliser. (ex : l'API fournit la fenêtre de base, mais on peut définir une fenêtre "éditeur" qu'on va instancier à volonté)
- Les composants peuvent capter des **évènements** (validation, clic souris, entrée clavier, redimensionnement, ...), dont le traitement est délégué à des fonctions de rappel (**gestionnaires d'évènements**) → **programmation évènementielle**

Compléments en POO
Aldric Degorre
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Concurrence
Interfaces graphiques
Interfaces graphiques
Principes
JavaFX
Stratégies
Gestion des erreurs et exceptions

Construire une GUI : mode d'emploi (1)

Pour une fenêtre "statique" :

1. Conceptualisez d'abord la fenêtre de votre application, dessin à l'appui.
2. Déterminez ses composants et leur hiérarchie (qui contient qui ?) sous forme d'arbre.
3. Programmez/écrivez [1] la description de la fenêtre, de ses composants et de leurs propriétés (taille, couleur, etc.) et des relations entre composants (notamment relations contentant/contenu).

À ce stade, votre programme peut afficher votre jolie fenêtre... qui ne fera rien [2]. Pour en faire une application utile, il faut maintenant associer des actions aux évènements.

1. En fonction du contexte, ça peut être un programme Java... ou bien un fichier dans un langage descriptif tel que HTML ou FXML.
2. On a en fait implémenté la partie "Vue" du patron MVC.

Construire une GUI : mode d'emploi (2)

Gérer les évènements

Pour gérer les évènements [1] **:**

1. On crée des **gestionnaires d'évènement**, spécifiques à chaque type d'évènement.
2. Pour chaque type d'évènement qu'on veut traiter sur un composant donné, on lui associe un gestionnaire, en le passant à une certaine méthode de ce composant.
   (Ceci peut se faire lors de l'initialisation du composant en question.)
3. Désormais, à chaque fois que cet évènement se produira, le gestionnaire sera exécuté avec pour paramètre un **descripteur d'évènement**.

Un gestionnaire d'évènement est une "fonction" [2] dont le paramètre est un **descripteur d'évènement** (de type souvent nommé `XXXEvent`), contenant la description des circonstances de l'évènement (composant d'origine, coordonnées, bouton cliqué ...).

1. Ceci correspond à la partie "Contrôleur" du patron MVC.
2. Fonction de rappel, matérialisée comme un objet contenant un méthode qui décrit la fonction ; c'est donc une fonction de première classe.

JavaFX

- A existé tantôt comme bibiliothèque séparée, tantôt comme composant du JDK (Java 8 à 10). À partir de Java 11, développé au sein du projet séparé OpenJFX [1].
- Avant Java 8, JavaFX pouvait être programmé via un langage de script appelé JavaFX Script, celui-ci a été abandonné depuis.
- JavaFX : bien plus qu'une bibiliothèque de description d'IG.
  Par exemple, en plus des composants typiques, on peut insérer dans l'arbre des formes 3D, et appliquer au tout diverses transformations géométriques, y compris en 3D.

Une particularité de JavaFX, c'est la possiblitié de décrire une IG via un langage de description appelé **FXML** (inspiré de HTML), et de définir le style des composants via des pages de style **CSS**. [2]

1. Mais certaines distributions de Java incluent JavaFX : citons Zulu de Microsoft et Liberica de Bellsoft.
2. Ce cours n'explique pas la syntaxe de FXML et de CSS, mais seulement de la construction de l'IG via des méthodes purement Java. Cependant, vous pouvez les utiliser en TP ou en projet.

Structure d'une IG en JavaFX

En JavaFX, nous avons la hiérarchie suivante :

- L'arbre est appelé **graphe de scène** (*scene graph*) et est constitué de **nœuds**, instances de sous-classes de `Node`.
- Les nœuds internes sont instances de sous-classes de `Parent`.
- Le nœud racine de l'arbre est associé à un objet de classe `Scene`. La **scène** correspond à la totalité de l'IG destinée à effectuer une tâche donnée.
- La scène, pour être affichée, doit être donnée à un objet de classe `Stage` [1] (le lieu où sera dessinée l'IG ; p. ex., sur un ordinateur de bureau : une fenêtre).

Cette organisation permet de facilement changer le contenu entier d'une fenêtre pour passer d'une tâche à l'autre : il suffit de dire au *stage* d'afficher une autre *scene*.

1. *scene* et *stage* : les deux se traduisent en Français par "scène" mais ont un sens très différent. *Scene* désigne une subdivision temporelle (scène = chapitre d'une pièce de théâtre), alors que *stage* désigne un lieu (scène = les planches sur lesquelles on joue la pièce).
Ainsi, pour éviter les confusions, soit je ne traduirai pas *stage* soit je dirai juste... une fenêtre !

Exécution de l'IG en JavaFX

1. Pour des raisons techniques [1], la construction ne peut être faite directement depuis `main()` où une méthode appelée par `main()`.
2. À la place, on doit créer une classe `MonAppJFX extends Application`, pour laquelle il faudra implémenter la méthode `void start(Stage stage)`.
   C'est dans cette méthode qu'on initie la construction.
3. Pour démarrer l'interface graphique (par exemple depuis `main()`) on fait : [2]

```
Application.launch(MonAppJFX.class);
```

ou bien, dans le cas où on fait l'appel depuis `MonAppJFX`, juste :

```
Application.launch();
```

1. Des histoires de *threads* dont nous reparlons juste après.
Cette contrainte n'est pas spécifique à JavaFX, mais inhérente à la programmation évènementielle.
2. Cette instruction lance la méthode `start()` dans le *thread* des évènements JavaFX.

Gérer les évènements en JavaFX

Pour gérer les évènements,

1. les gestionnaires d'évènement de JavaFX implémentent l'interface `EventHandler<T>` :

```
public interface EventHandler<T extends Event> {
  void handle (T e);
}
```

   (où `T` peut être remplacé par le type d'évènement à traiter, ex : `ActionEvent`, `KeyEvent`, `MouseEvent`, ...).

2. on associe un gestionnaire d'évènement à un composant JavaFX ainsi : `composant.setOnXXXX(gestionnaire)` (ex : `void setOnMousePressed(EventHandler<? super MouseEvent> gest)`)
3. à partir de désormais, quand un évènement du type indiqué se produit, la méthode `handle()` du gestionnaire est exécutée.

Gérer les évènements en JavaFX

exemple

Prenons l'exemple d'`ActionEvent`. Je peux par exemple créer la classe :

```
public class GererEnregistrement implements EventHandler<ActionEvent> {
  private final Document doc;
  public GererEnregistrement(Document d) { this.doc = d; }
  public void handle(ActionEvent e) { d.enregistre(); }
}
```

Supposons maintenant que la variable `boutonEnregistrer` désigne un composant de type `Button`, alors pour que cliquer sur le bouton désormais déclenche l'enregistrement, il suffit d'ajouter l'instruction :

```
boutonEnregistrer.setOnAction(new GererEnregistrement(documentCourant));
```

Remarque : si `e` objet évènement, alors `e.getSource()` référence le composant où l'évènement a été créé. Cette référence peut servir faire un traitement différencié en fonction de l'état du composant.

Gérer les évènements

Utilisez les lambda-expressions !

`EventHandler` étant une interface fonctionnelle, lambda-expressions possibles :

Pour les gestionnaires courts, préférer les lambda-abstractions :

```
composant.setOnMousePressed(e -> { /* gérer l'évènement e ici */ });
```

Pour les gestionnaires longs, écrire un méthode et en passer une référence :

```
  void methodeDuClic(MouseEvent e) { /* gérer l'évènement e ici */ }
  ...
  composant.setOnMousePressed(controleur::methodeDuClic); // référence de méthode
```

Dernière technique intéressante si programme plus long qu'un simple exemple : la partie où on associe composants et gestionnaires est plus succincte et claire. En plus, on peut regrouper les méthodes de gestion d'événement dans dans une même classe [1].

1. le contrôleur du patron MVC

Exécution des évènements

Parler de *threads* sans en parler.

- Les méthodes `handle()` des gestionnaires d'évènement s'exécutent **les unes après les autres** (jamais en même temps).
- De même, ces gestionnaires commencent à s'exécuter seulement **après** la méthode `start()` de l'`Application` (et sa pile d'appels).
- En revanche, la méthode `main()` (et sa pile d'appels) continue à s'exécuter **en parallèle** → ne jamais tenter de modifier/ajouter un composant JavaFX depuis `main()` ou une méthode appelée par `main()` (résultats imprévisibles).
- **Remarque :** un traitement long [1] ou un appel bloquant dans un gestionnaire peut donc ralentir ou bloquer toute l'application.

  Si on a besoin d'exécuter du code long ou bloquant, il faudra le lancer **en parallèle** sur un autre *thread* (concept de ***worker thread*** [2]).

1. i.e. plus long que l'intervalle de rafraîchissement de la fenêtre
2. Vous pouvez, à cet effet, créer un *thread* classique ou, mieux, utiliser `javafx.concurrent.Worker`, l'API prévue par JavaFX, ou bien toute autre API de votre convenance.

Threads en JavaFX

Pour parler "*threads*", ainsi 3 sortes de *threads* dans un programme JavaFX :

- 1 *thread* initial (`main`) : dans une application JavaFX, ne sert qu'à démarrer le JFXAT, via l'appel à `Application.launch()` en plaçant un évènement initial (l'exécution de `start()`) dans la file d'attente des évènements.
- 1 *thread* d'application JavaFX (JFXAT) qui lance les tâches de sa file d'attente (typiquement, les gestionnaires d'évènement).
  - les gestionnaires d'événements sont automatiquement programmés sur le JFXAT
  - ajout possible de tâches depuis autre *thread* avec `Platform.runLater(task)`.
  - Intérêt du JFXAT : permettre à l'IG de tourner indépendamment du reste du programme, en restant réactive.
  - Pourquoi restreindre toute manipulation de l'IG à ce seul JFXAT : on évite les problèmes usuels [1] du *multithreading* en rendant les exécutions de gestionnaires atomiques [2] les unes par rapport aux autres.
- parfois des ***worker threads*** pour les tâches longues ou bloquantes.

1. Entrelacements indésirables, accès en compétition, ...(cf. chapitre sur la concurrence)
2. Elles ne s'interrompent pas les unes les autres.

Exemple JavaFX minimaliste

```
import javafx.application.Application; import javafx.scene.*; import
    javafx.scene.control.*; import javafx.scene.layout.BorderPane;

public class JFXSample extends Application {
    @Override public void start(Stage stage) throws Exception {
        MenuItem exit = new MenuItem("Exit"); exit.setOnAction(e -> System.exit(0));
        Menu file = new Menu("File"); file.getItems().add(exit);
        MenuBar menu = new MenuBar(); menu.getMenus().add(file);
        Label lbl = new Label("Exemple de Label qui tourne...");
        lbl.setOnMousePressed(e -> lbl.setRotate(lbl.getRotate() + 10));
        BorderPane root = new BorderPane(); root.setTop(menu); root.setCenter(lbl);
        Scene scene = new Scene(root, 400, 400);
        stage.setScene(scene);            // définir la scène à  afficher
        stage.setTitle("Application test");
        stage.show();                              // lancer l'affichage !
    }

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


Composants de JavaFX

- Vous en avez vus quelques uns dans l'exemple précédent.
- Pour plus d'exemples, le mieux, c'est d'ouvrir les tutoriels que l'on peut trouver sur le web.
- Sinon, vous trouverez une liste exhaustive en regardant la documentation du *package* `javafx.scene.control` : `https://openjfx.io/javadoc/11/javafx.controls/javafx/scene/control/package-summary.html`

Attention quand vous trouvez de la documentation : vérifiez qu'elle concerne bien la version installée chez vous. [1].

1. Les moteurs de recherche tendent encore trop à référencer d'anciennes versions comme JavaFX 2...

Organiser une application graphique

Séparation vue et modèle

C'est souvent une bonne idée de séparer les deux aspects suivants :

- **Modèle :** cœur du programme, partie « métier ». C'est ici que sont gérées, organisées, traitées les données. On y trouve les déclarations de structures de données ainsi que les méthodes implémentant les différents algorithmes traitant sur ces structures.
- **Vue :** partie qui sert à présenter l'application à l'utilisateur et sur laquelle l'utilisateur agit.

Idéalement, les classes du modèle **ne dépendent pas** des classes de la vue [1].

Plusieurs stratégies pour coordonner M et V, notamment : Model-View-Controller (MVC), Model-View-Presenter (MVP) et Model-View-ViewModel (MVVM) (dans les 3 cas : ajout d'un 3e composant).

1. ce qui permet de changer la présentation de l'application, notamment pour la porter sur des plateformes différentes

Organiser une application graphique

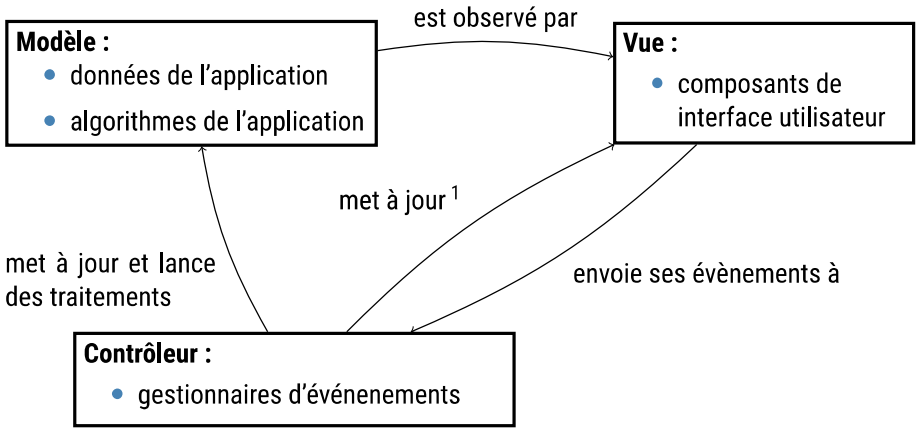
L'architecture MVC

Architecture MVC décrite depuis 1978... encore très populaire pour les applications graphiques, notamment les applications web. Le troisième composant est le **contrôleur**.

- **Modèle :** déjà décrit.
- **Vue :** déjà décrite.
- **Contrôleur :** partie du programme servant à interpréter les évènements (entrées de l'utilisateur dans la vue, mais pas seulement) pour agir sur le modèle (déclencher un traitement, ...) et la vue (ouvrir un dialogue, ...).

JavaFX est particulièrement adapté à mettre en œuvre une stratégie MVC.

Illustration



1. pour la partie de l'interface utilisateur qui ne sert pas à la présentation des données, par exemple : ouverture des menus, boîtes de dialogue, etc.

Le patron de conception Observateur/Observable

En général

- **Cas d'application :** quand les changements d'état d'un objet donné (l'**observable**) peut avoir des répercussions sur de multiples autres objets (les **observateurs**).
- **Principe :** Chaque observable contient une liste d'observateurs abonnés. Quand un changement a lieu, il appelle sur chaque élément de cette liste une même méthode (d'une interface commune) pour prévenir tous ses observateurs.
- **Intérêt :** éviter que la classe de cet objet ne dépende des classes de ses observeurs (la dépendance se créée entre objets, à l'exécution).
- Patron utilisé pour la relation Modèle $\leftrightarrow$ Vue dans MVC : V réagit aux changements de M, sans que celui-ci n'ait de dépendance vers la seconde.

Le patron de conception Observateur/Observable

Avec `java.util`

Implémentation "historique" de Java : interface `java.util.Observer` et classe `java.util.Observable`. Regardez leurs documentations.

En plus de ce qu'on vient de décrire, `java.util.Observable` contient un mécanisme pour éviter de notifier tout le temps les observateurs, via le couple de méthodes `setChanged()`/`hasChanged()` :

- `setChanged()` : sert à signaler que l'observable vient d'être modifié ;
- `hasChanged()` : renvoie `true` si l'observable a été modifié depuis la dernière notification aux observateurs.

Le patron de conception Observateur/Observable

Avec `javafx.beans` (1)

JavaFX a sa propre implémentation de ce patron.

Les **propriétés** des composants graphiques (mais pas seulement) sont matérialisées par des instances de `javafx.beans.Property`, sous-interface de `javafx.beans.Observable`.

Pour toute propriété de nom “`value`”, le composant contiendra les 3 méthodes suivantes :

- `public Double getValue()` : lire la valeur de la propriété
- `public void setValue(Double value)` : modifier la valeur de la propriété
- `public DoubleProperty valueProperty()` : obtenir l’objet-propriété lui-même

Compléments en POO
Aldric Degorre
Introduction
Généralités
Style
Objets et classes
Types et polymorphisme
Héritage
Généricité
Concurrence
Interfaces graphiques
Interfaces graphiques
Principes
JavaFX
Stratégies
Gestion des erreurs et exceptions

Le patron de conception Observateur/Observable

Avec `javafx.beans` (2))

Comme ces propriétés sont observables, on peut leur ajouter des observateurs :

```
slider.valueProperty().addListener(
    (observable, oldVal, newVal) -> afficheur.textProperty().setValue("valeur : " +
        newVal));
```

Remarque : les composants graphiques contiennent de telles propriétés (cf. exemple), mais elles peuvent aussi être utilisées dans toute autre classe, notamment dans la partie Modèle de l’application, permettant l’observation du Modèle par la Vue.