Génie Logiciel Avancé : interface graphique en JavaFX

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

5 février 2025

amU Faculté des sciences Aix Marseille Université

Section 1

Introduction JavaFX

Historique

- Les premières IHM en Java se faisaient en utilisant la bibliothèque AWT : Composants “lourds” dessinés par le système (dont l’apparence est donc très dépendante du système)
- Rapidement, la bibliothèque Swing est venue remplacer AWT : composants “légers” dessinés par la bibliothèque (en Java)
- JavaFX 1 tentative ratée de remplacer Swing : basé sur un langage de script (JavaFX Script)
- Une refonte importante du *toolkit* a pris en compte les critiques formulées et a conduit à une nouvelle mouture : JavaFX 2
- Caractéristiques principales :
  - Abandon du langage de script
  - Possibilité d’utiliser l’API et/ou un langage descriptif (syntaxe XML)
  - Possibilité de génération de FXML via l’outil interactif Scene Builder
  - Utilisation possible de feuilles de styles CSS

Qu'est-ce qu'est JavaFX ?

JavaFX

framework pour créer des interfaces graphique pour des applications de bureau ou smartphones.

- Première version en 2008
- successeur de `Swing`
- Créé et développé par Sun → Oracle → OpenJFX
- Inclus avec JDK 8 à 10, disponible séparément pour JDK 11+

HelloWorld version JavaFX

```java
import javafx.application.Application;
import javafx.stage.Stage;

public class HelloApplication extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("Hello world !");
    primaryStage.show();
  }
  public static void main(String[] args) {
    Application.launch(args); // méthode qui exécutera start
  }
}
```

Explication `HelloApplication`

La classe principale est toujours une classe étendant `Application`.

`Application.launch(args)` permet de lancer l'application avec des arguments qui peuvent être récupérées avec `getParameters()` (méthode d'`Application`)

Le runtime *JavaFX* réalise les étapes suivantes lorsqu'une `Application` est lancée :

1. Instancie l'instance de type `Application`
2. Appelle `init()` (méthode d'`Application`)
3. Appelle `start(stage)` (méthode d'`Application`) avec `stage` le contenu d'une fenêtre par défaut de type `Stage`
4. Attends la fin de l'exécution de l'application (`Platform.exit()` ou fermeture des fenêtres)
5. Appelle la méthode `stop()` (méthode d'`Application`)

JavaFX avec gradle (1/2)

Gradle

Moteur de production : outil pour faciliter la gestion de dépendances, la compilation, la génération de code, ...

Pour javaFX, il existe un plugin pour javafx (fichier `build.gradle`) :

```
plugins {
    id 'org.openjfx.javafxplugin' version '0.1.0'
}
javafx {
    version = '23.0.2'
    modules = ['javafx.controls']
}
```

JavaFX avec gradle (2/2)

Obligation de passer par `gradle run` pour lancer l'application via le plugin application :

```
plugins {
    id 'java'
    id 'application'
}
application {
    mainModule = 'fr.univ_amu.m1info.gla_javafx'
    mainClass = 'fr.univ_amu.m1info.gla_javafx.HelloApplication'
}
```

Demande de définir un module java.

Modules en Java

- Existe depuis Java 9+
- Permet de regrouper des *package*
- Permet de spécifier les dépendances entre module et les parties du module visible depuis l'extérieur
- Défini via un fichier `module-info.java` à la racine du module

Exemple de fichier `module-info.java` :

```
module fr.univ_amu.m1info.gla_javafx {
    requires javafx.controls;
    exports fr.univ_amu.m1info.gla_javafx;
    opens fr.univ_amu.m1info.gla_javafx to javafx.fxml;
}
```

Format fichier `module-info.java`

Commence par `module nom.du.package` puis des lignes indiquant les dépendances et les parties ouvertes du module :

- `requires other.module` : indique une dépendance à `other.module`
- `exports package` : rend accessible les classes publiques de `package` aux autres modules
- `opens package to module` : ouvre à l'introspection les membres non-publiques des classes du *package* au *module*
- `uses class` : indique que le module utilise la classe `class`

Introspection en Java API Reflection

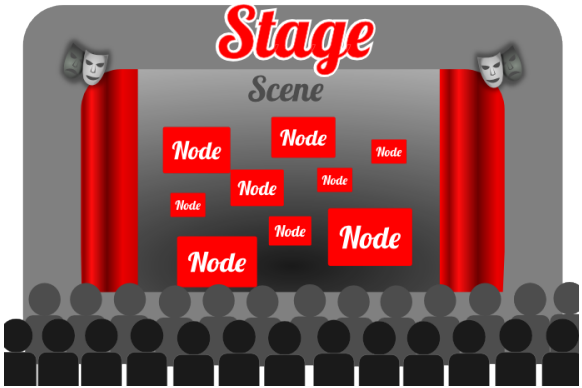
Java API `Reflection` : permettant d'inspecter les classes, les attributs, les méthodes et les constructeurs ainsi que leurs annotations.

- `clazz = new Person().getClass()` ou `clazz=Class.forName("NomDeClasse")` pour récupérer une classe
- `clazz.getDeclaredFields()` pour récupérer les attributs
- `clazz.getMethods()` pour récupérer les méthodes
- `clazz.getModifiers()` pour récupérer le type de la classe
- `getInterfaces()` pour récupérer les interfaces
- `getAnnotationsByType()` récupère les annotations associées sur un attribut ou une méthode pour
- possibilité de rendre un attribut ou une méthode accessible (et donc utilisable même si privé)

Section 2

Base de JavaFX

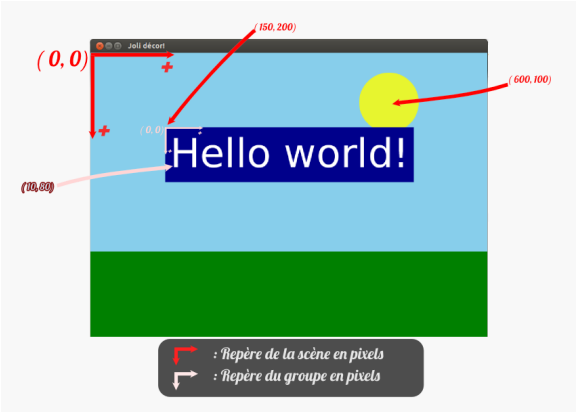
Éléments de JavaFX : terminologie du théâtre

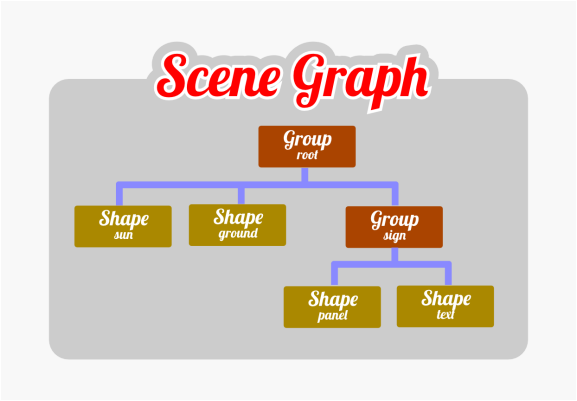


- `Stage` : estrade correspondant à la fenêtre (système de fenêtre des systèmes d'exploitation)
- `Scene` : décor où a lieu l'action (possibilité d'avoir plusieurs décors pour une même estrade)
- `Node` : nœuds placés dans le décor et donc correspondant aux acteurs

source image : https://mikarber.developpez.com/tutoriels/java/introduction-javafx/

Arbre d'une scène

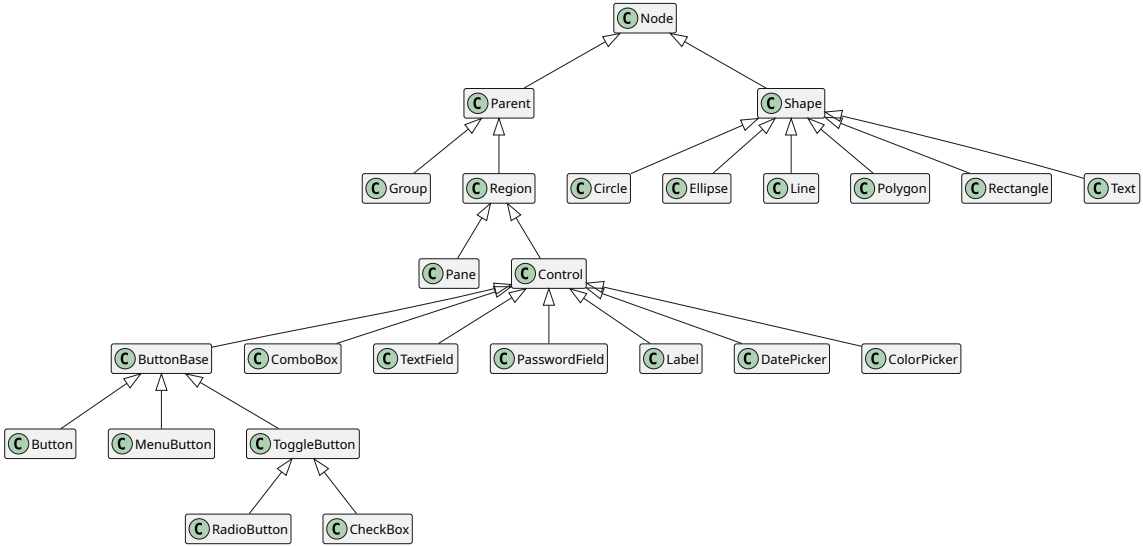




Les éléments d'une scène sont organisés sous forme d'un arbre :

- Des nœuds internes containeurs de type `Parent extends Node`
- Des nœuds feuilles de type `Node` : boutons, champs de saisie ou texte, formes, ...

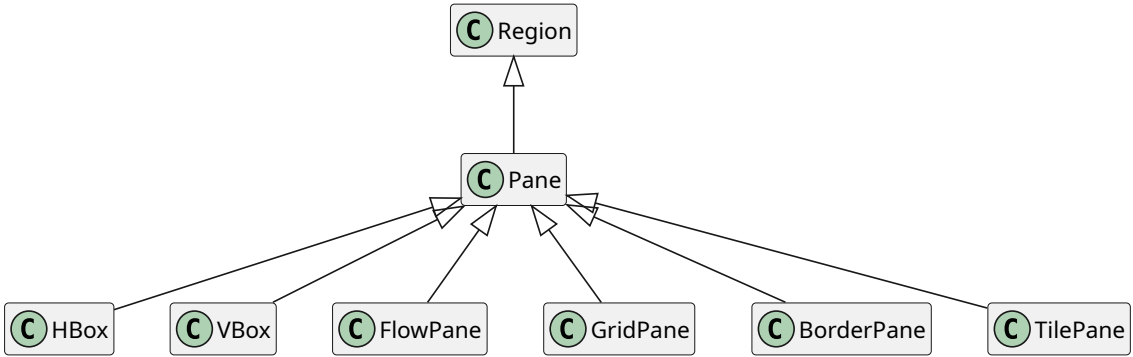
La classe `Node` et ses sous-classes



Classes conteneurs

Conteneurs (*Layout*)

Nœuds qui permettent d'indiquer l'organisation des composants sur la scène



Conteneurs : HBox et VBox

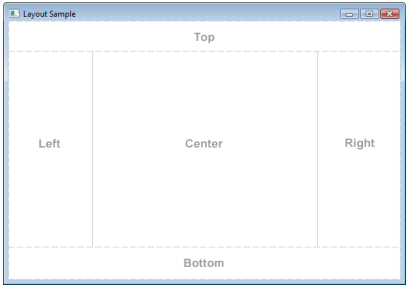
- HBox :
  - placement des composants sur une ligne horizontale, de gauche à droite
  - des fonctions permettent d'adapter le conteneur : `setAlignment()`, `setMinWidth()`, `setSpacing()`, ...
- VBox
  - idem que `HBox` mais en vertical

```
public void start(Stage primaryStage) throws Exception {
  Label label = new Label("Salut le monde !");
  Button bye = new Button("Bye");
  VBox vBox = new VBox();
  vBox.getChildren().addAll(label, bye);//ajout des nœuds à la suite
  Scene scene = new Scene(vBox);
  primaryStage.setScene(scene);
  primaryStage.show();
```

Conteneurs : BorderPane

5 zones : Haut, Bas, Gauche, Centre, Droite

- un seul nœud par zone
- le nœud du centre aura la tendance d'occuper le plus de place
- découpage classique d'une fenêtre




```
BorderPane root = new BorderPane();
root.setLeft(new Button("Left"));
root.setRight(new Button("Right"));
root.setTop(new Button("Top"););
root.setBottom(new Button("Bottom"));
root.setCenter(new Button("Center"));
Scene scene = new Scene(root);
```

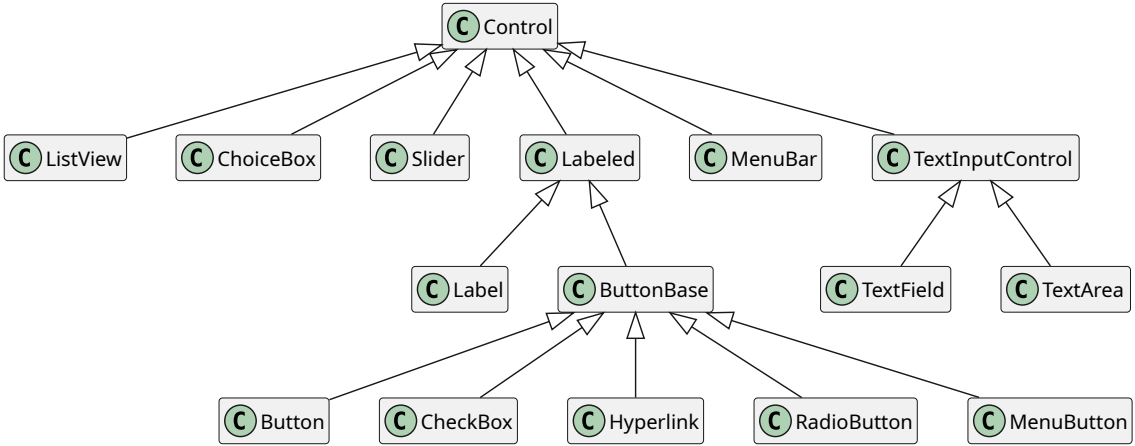
Conteneurs : TilePane, GridPane, FlowPane et StackPane

Permettent d'indiquer l'organisation des composants :

- `TilePane`
  - placement sous forme d'une grille
  - toutes les cases de la grille ont la même taille
- `GridPane`
  - placement sous forme d'une grille
  - les lignes/colonnes peuvent être de taille variable
- `FlowPane`
  - les éléments sont disposés sur une ligne (horizontale ou verticale)
  - lorsque il n'y a plus assez de place disponible, on passe à la ligne suivante
- `StackPane`
  - les composants sont organisés sous forme d'une pile (seul le sommet de la pile est visible)
  - exemple : une pile de cartes dans un jeu

Composants graphiques (visibles)

Tous les composants interagissant directement avec l'utilisateur héritent de la classe abstraite `javafx.controls.Control`



Quelques composants de base : Label

Une simple étiquette affichée (texte, icône), non-éditable

```
Label etiquette = new Label("Je suis un label textuel");
etiquette.setFont(Font.font("Cambria", 18));
etiquette.setTextFill(Color.DARKCYAN);

InputStream imageStream =
  ClassLoader.getSystemResourceAsStream("JavaFXLogo.png");
Image image = new Image(Objects.requireNonNull(imageStream));
Label labelAvecImage = new Label();
labelAvecImage.setGraphic(new ImageView(image));
root.setCenter(labelAvecImage);
```

Le fichier `JavaFXLogo.png` doit être dans le répertoire `src/main/resources`

Quelques composants de base : TextField (1/2)

- permet de créer un champ de saisie (une ligne)
- possibilité d'associer un traitement (par ex. en fonction du texte saisi)

```
VBox root = new VBox();
Label message = new Label("Tapez ici vos secrets :");
TextField champ = new TextField();
// personnalisation du champ de texte
champ.setMaxWidth(260);
champ.setText("Je suis un extraterrestre");
// ajout des 2 noeuds au conteneur
root.getChildren().addAll(message, champ);
```

Quelques composants de base : TextField (2/2)

On peut associer un traitement grâce (en autre) à `setOnAction` qui a un argument `EventHandler<ActionEvent>`.

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

Il est possible de récupérer l'objet source de l'événement :

```
field.setOnAction(event -> {
    TextField source = (TextField) event.getSource();
    System.out.println(source.getText());
  });
```

Quelques composants de base : Button

Trois types de boutons :

- exécution de commandes (`Button`, `Hyperlink` et `MenuButton`)
- pour faire des choix (`ToggleButton`, `CheckBox` et `RadioButton`)
- combinaison des deux (`SplitMenuButton`)

Tous les boutons permettent un traitement lors du clic

```
Button bouton = new Button("Joli bouton");
//le clic sur le bouton provoque un traitement (ici affichage) :
bouton.setOnAction(e -> System.out.println("clic intercepté"));
```

Section 3

Gestion des événements

Introduction à la programmation événementielle

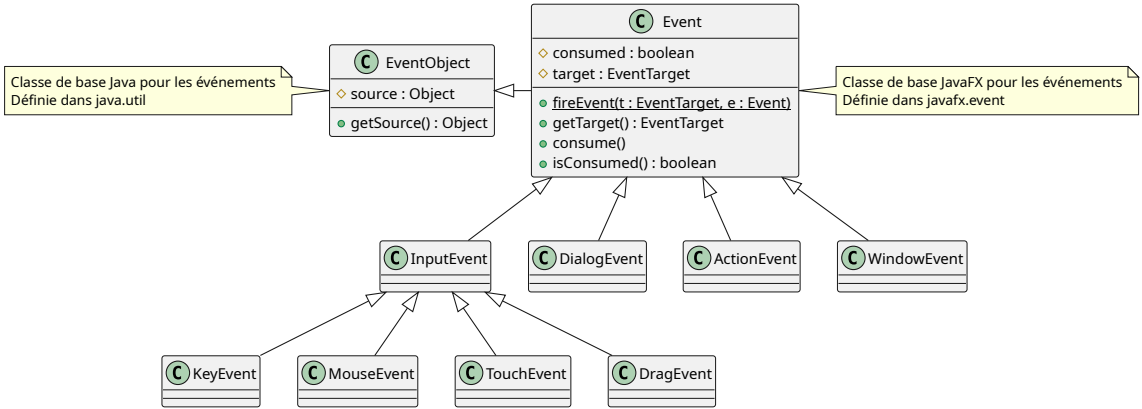
Evénement

Une action qui peut être identifiée par un programme informatique et qui peut être gérée par le programme grâce à un gestionnaire d'événements (`EventHandler` en Java).

Pour JavaFX, un événement est :

- généralement associé à une action de l'utilisateur
- une instance de `javafx.event.Event` ou d'une de ses sous-classes (dont `MouseEvent`, `KeyEvent`, `DragEvent`, `ScrollEvent`,...)
- associé à une source, une cible et un type d'événement

Hiérarchie (partielle) des classes d'événements JavaFX



Les événements en JavaFX

Tous les `Event` de JavaFX ont 3 propriétés :

1. Une *source* un `Object` Java qui a généré l'événement
2. Une destination (`EventTarget`) : typiquement un élément du graphe de scène ou la fenêtre principale
3. Un type d'événement : spécifie l'événement concret sous-jacent

Principe de fonctionnement :

- Lorsqu'un événement a lieu, souvent on souhaite définir une réponse de réaction sur l'interface graphique.
- Pour cela il faut enregistrer un gestionnaire d'événement (type `EventHandler`) sur l'élément source.
- Lorsque l'événement est détecté par l'environnement Java, celui-ci exécutera le code correspondant défini dans l'objet `EventHandler` correspondant.

Interface EventHandler

```
// une interface fonctionnelle (une et une seule méthode abstraite)
public interface EventHandler<T extends Event> extends EventListener
  void handle(T var1);
}
```

- la méthode reçoit un événement JavaFX
- l'intégralité du code de réaction au déclenchement de l'événement y est défini
- instanciable via une lambda expression

Exemple d'enregistrement de *listener*

```
EventHandler<ActionEvent> handler = e -> System.out.println("test");
scene.addEventHandler(MouseEvent.MOUSE_CLICKED, handler);
```

Remarques sur l'enregistrement (1/2)

Les *handlers* (gestionnaires) peuvent être enregistrés sur tous les objets ayant la méthode `addEventHandler(...)` :

```
// signature de la méthode d'enregistrement d'un
// gestionnaire d'événement
<T extends Event> void addEventHandler(EventType<T> type,
                    EventHandler<? super T> gestionnaire)
```

- Le type générique `T` désigne le type d'événement (par ex. `MouseEvent`)
- Le paramètre type de la fonction permet de préciser le type concret d'événements :
  - pour `KeyEvent` : `KEY_PRESSED`, `KEY_RELEASED`, `KEY_RELEASED`, ...
  - pour `MouseEvent` : `MOUSE_PRESSED`, `MOUSE_RELEASED`, `MOUSE_CLICKED`, `MOUSE_MOVED`, ...

Les classes `Node`, `Scene` définissent une méthode `addEventHandler(...)`

Remarques sur l'enregistrement (2/2)

Il est possible d'ajouter plusieurs *handlers* sur le même nœud :

```
Button btn = new Button("Hello");
Rectangle rectangle = new Rectangle(80, 120);
rectangle.setFill(Color.RED);
btn.addEventHandler(MouseEvent.MOUSE_CLICKED,
                    _ -> btn.setText("click"));
btn.addEventHandler(MouseEvent.MOUSE_MOVED, _ -> {
  if (rectangle.getFill() == Color.RED)
    rectangle.setFill(Color.GREEN);
  else
    rectangle.setFill(Color.RED);
});
```

Les sources et les cibles des événements

- La méthode `getSource()` de `Event` renvoie la source de l'événement :

  le nœud sur lequel `addEventHandler(...)` a été invoquée
- La méthode `getTarget()` de `Event` renvoie la cible de l'événement :

  le nœud sur le graphe de scène sur lequel l'événement va agir
- en JavaFX souvent la source et la cible sont les mêmes, mais pas toujours !

Exemple de sources et cibles (1/2)

```
Circle cercle = new Circle(250, 250, 250);
HBox racine = new HBox();
racine.getChildren().add(cercle);
Scene scene = new Scene(racine);
EventHandler<MouseEvent> gestionnaire = mouseEvent -> {
  System.out.println("Source :"
    + mouseEvent.getSource().getClass().getSimpleName());
  System.out.println("Cible :"
    + mouseEvent.getTarget().getClass().getSimpleName());
};
// enregistrement du même gestionnaire sur plusieurs sources
scene.addEventHandler(MouseEvent.MOUSE_CLICKED, gestionnaire);
racine.addEventHandler(MouseEvent.MOUSE_CLICKED, gestionnaire);
cercle.addEventHandler(MouseEvent.MOUSE_CLICKED, gestionnaire);
```

Exemple de sources et cibles (2/2)

Un clic de souris sur le cercle provoque l'affichage suivant :

```
Source :Circle
Cible :Circle
Source :HBox
Cible :Circle
Source :Scene
Cible :Circle
```

Distinction `Source`/`Target`

- `Source` : objet ayant capturé l'*event*
- `Target` : objet sur lequel s'applique l'*event* (par exemple l'objet au premier plan sur la position du curseur dans le cas d'un `MouseEvent`)

Remarques

- les lambdas sont très pratiques, mais à éviter lorsque le même gestionnaire est à utiliser sur plusieurs éléments
- Pour supprimer un gestionnaire :

```
Bouton bouton = new Bouton("Coucou");
// création d'un événement
EventHandler<MouseEvent> gestionnaireSouris =
  evenement -> System.out.println("clické !");
// ajout du gestionnaire
bouton.addEventHandler(MouseEvent.MOUSE_CLICKED, gestionnaireSouris);
// suppression du gestionnaire
bouton.removeEventHandler(MouseEvent.MOUSE_CLICKED,
                          gestionnaireSouris);
```

Les événements en JavaFX : méthodes de convenances

Pour les événements les plus courants, l'enregistrement des gestionnaires peut être simplifié :

```
// création d'un événement
EventHandler<MouseEvent> mouseEventHandler =
evenement -> System.out.println("événement de souris");
// ajout de l'écouteur à la scène avec une méthode de convenance
scene.setOnMouseClicked(mouseEventHandler);
```

- les méthodes `setOnXXX()` (par exemple : `setOnKeyTyped()`, `setOnMouseClicked()`, ...) sont dites de convenances
- ne permettent pas d'attacher plusieurs gestionnaires
- existent uniquement pour les événements les plus courants pour un type de `Node` donné

Section 4

Propriétés et *bindings*

Contexte

Rappel - Encapsulation

Les accès à l'état interne de l'objet ne doivent pas (en principe) être exposés.

- les attributs doivent généralement être privés si besoin d'accéder à un attribut, alors il faut passer par une méthode publique

Convention JavaBeans ($\neq$ Enterprise JavaBeans)

- Tous les attributs privés d'une classe doivent être accessibles publiquement via des *getters* et modifiables grâce des *setters*.
- Constructeur sans arguments

Si un objet respecte la convention JavaBeans, alors il est possible de le rendre observable et de lui associer des *listener*.

Les propriétés

Définition d'une Property

Une propriété est un élément d'une classe que l'on peut manipuler à l'aide de *getters* et de *setters* (couple *getter*/*setter*).

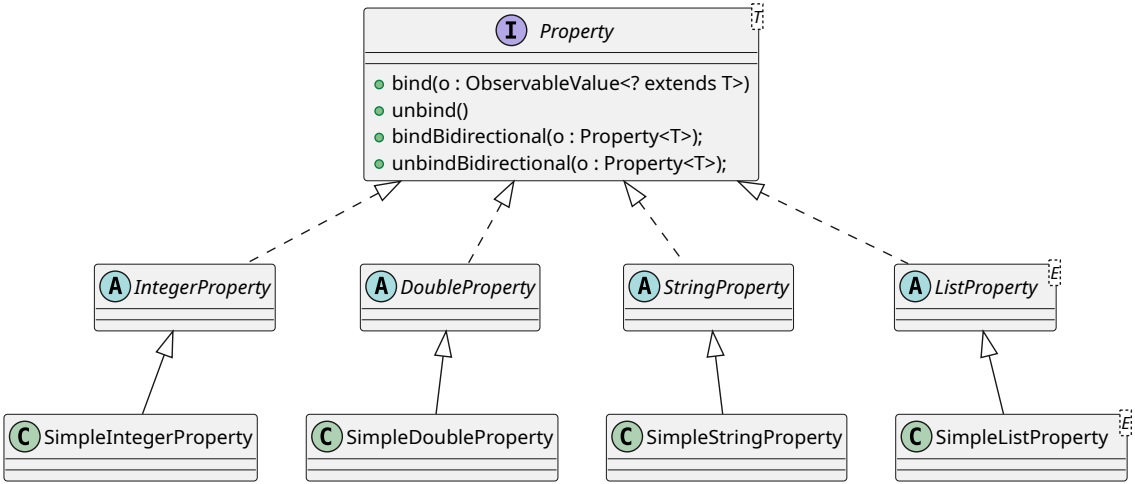
Les propriétés en JavaFX :

- Peuvent représenter une valeur ou un ensemble de valeurs
- Peuvent être en écriture et/ou en lecture
- En plus des méthodes `getXXX()` et `setXXX()`, les propriétés possèdent la méthode `XXXProperty()` qui retourne un objet qui implémente l'interface `Property`

Exemple de propriété

```
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
public class Book {
  private StringProperty title = new SimpleStringProperty("titreIniti
  public final String getTitle() {
   return title.get();
  }
  public final StringProperty titreProperty() {
    return title;
  }
  public final void setTitle(String title) {
    this.titre.set(titre);
  }
}
```

Properties en JavaFX



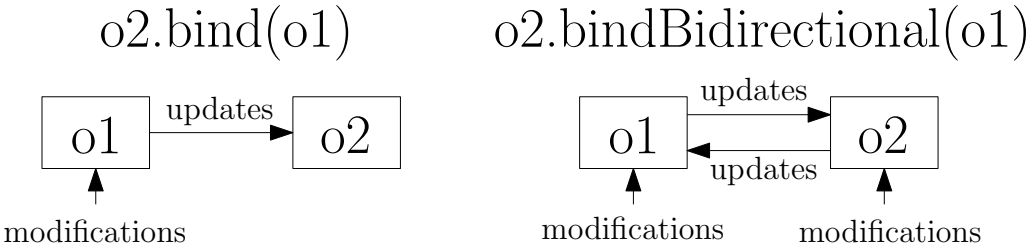
Bindings

Binding unidirectionnel (`o2.bind(o1)`)

Changement de `o1` ⇒ changement `o2` (impossible de `set o2`).

Binding bidirectionnel (`o2.bindBidirectional(o1)`)

Les changements de `o1` sont répercuté sur `o2` et inversement.

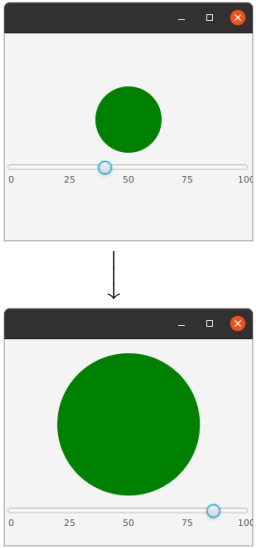


Exemple de bind unidirectionnel

```
BorderPane root = new BorderPane();
root.setMinWidth(250);
root.setMinHeight(300);

Slider slider = new Slider();
slider.setValue(40);
slider.setShowTickLabels(true);
root.setBottom(slider);

Circle cercle = new Circle();
cercle.setFill(Color.GREEN);
root.setCenter(cercle);
cercle.radiusProperty().bind(slider.valueProperty());
```



Exemple de bind bididirectionnel

```
IntegerProperty propA = new SimpleIntegerProperty(10);
IntegerProperty propB = new SimpleIntegerProperty(32);
// création d'un lien bidirectionnel
propB.bindBidirectional(propA);
System.out.println(propA.getValue()); // affiche 10
System.out.println(propB.getValue()); // affiche 10
propA.setValue(5);
System.out.println(propA.getValue()); // affiche 5
System.out.println(propB.getValue()); // affiche 5
propB.setValue(8);
System.out.println(propA.getValue()); // affiche 8
System.out.println(propB.getValue()); // affiche 8
```

Création des bindings haut niveau

La classe utilitaire `Bindings` permet de créer des liaisons haut niveau :

- calculs : `add`, `divide`, `multiply`, `max`, `min`, ...
- logique : `and`, `or`, èqual`,`lessThan`,`greaterThan‘, ...
- conversion en chaînes de caractères : `concat`, `convert`, ...
- expression ternaire : `when(condition).then(v1).otherwise(v2)`

Possibilité de chaîner les méthodes de *binding* (Fluent API) :

```
IntegerProperty number = new SimpleIntegerProperty(7);
StringBinding parity = new When(number.divide(2).multiply(2)
    .isEqualTo(number)).then("pair").otherwise("impair");
System.out.println(parity.getValue()); // impair
number.set(8);
System.out.println(parity.getValue()); // pair
```

Section 5

FXML

FXML langage pour décrire des `Node`

- FXML est un langage basé sur XML pour décrire des interfaces graphiques JavaFX.
- Possible de créer ces fichiers avec un éditeur de texte mais aussi via un outil graphique qui génère automatiquement le FXML
- Un fichier FXML est au format XML et décrit :
  - la structuration du graphe de scène,
  - les propriétés graphiques de chaque composant (taille, police, couleur, ...)
- Le code FXML est modifiable indépendamment du code métier de l'application
- Le fichier FXML est chargé par l'application (classe `FXMLLoader`) et un objet Java (racine) est créé avec les éléments que le fichier décrit

Avantage principal

Une séparation nette entre le code métier et l'interface graphique utilisateur.

Interprétation des fichiers FXML

L'interprétation du contenu du fichier FXML crée des objets Java correspondants. Par exemple, l'élément :

```
<BorderPane prefHeight="80.0" prefWidth="250.0" . . .>
```

sera interprété comme :

```
BorderPane rootPane = new BorderPane();
rootPane.setPrefHeight(80.0); rootPane.setPrefWidth(250.0);
```

Un attribut avec le nom d'une classe suivi d'un point et d'un identificateur comme :

```
<TextField GridPane.columnIndex="3" . . . >
```

sera interprété comme une invocation de méthode statique :

```
TextField tfd = new TextField(); GridPane.setColumnIndex(tfd, 3);
```

Imbrication d'éléments

Certaines propriétés ne pouvant pas être représentées par une chaîne de caractères, on imbrique alors un élément

Par exemple, la propriété `Font` peut être un élément imbriqué dans l'élément Label

```
<Label id="title" fx:id="title" text="Titre" textFill="#0022cc"
    BorderPane.alignment="CENTER">
  <font>
    <Font name="SansSerif Bold" size="20.0" />
  </font>
</Label>
```

Pour les propriétés de type liste (par exemple `children`), les éléments de la liste sont simplement imbriqués et répétés dans l'élément représentant la liste (par exemple, les composants enfants seront listés entre les balises `<children>` et `</children>`)

Déclaration du contrôleur

Il est possible d'associer un contrôleur qui va gérer les interactions utilisateurs avec les éléments décrit dans le FXML

Cette classe doit être annoncée dans l'élément racine du fichier FXML, en utilisant l'attribut `fx:controller` :

```
<BorderPane prefHeight="80.0" prefWidth="250.0"
  style="-fx-background-color: #FFFCAA;"
  xmlns=http://javafx.com/javafx/8
  xmlns:fx=http://javafx.com/fxml/1
  fx:controller="SayHelloController">
  <children> ... </children>
</BorderPane>
```

Injection des composants dans le contrôleur

Les composants décrits dans le fichier FXML qui possèdent un attribut fx:id seront injectés dans le contrôleur en tant que variables d'instance :

```
<Button fx:id="btnHello" onAction="#handleButtonAction">
<Label id="title" fx:id="title" text="Titre" textFill="#0022cc" ...>
```

L'attribut fx:id fonctionne en lien avec l'annotation `@FXML` dans le code du contrôleur, qui précède la déclaration de la variable d'instance :

```
public class SayHelloController {
   @FXML  // fx:id="btnHello"
   private Button btnHello; // Object injected by FXMLLoader
   @FXML // fx:id="title"
   private Label title; // Object injected by FXMLLoader
}
```

Définition des actions associées

Pour les composants déclarés en FXML qui peuvent réagir à des événements, on peut indiquer la méthode du contrôleur, qui définit l'action à exécuter lorsque l'événement se produit, en utilisant l'attribut `fx:onEvent="#methodName"` :

```
<Button fx:id="btnHello" onAction="#handleButtonAction"
        text="Say Hello" BorderPane.alignment="CENTER" />
```

Dans la classe contrôleur, ces méthodes devront (comme les composants associés) être annotées avec @FXML

```
@FXML
private void handleButtonAction(ActionEvent event) {
   title.setText("Hello !");
   title.setTextFill(Color.FUCHSIA);
}
```

Initialisation du contrôleur

- Dans les classes qui agissent comme “contrôleurs”, on peut définir une méthode `initialize()` (qui doit être annotée avec `@FXML`) pour effectuer certaines initialisations
- Cette méthode est automatiquement invoquée après le chargement du fichier FXML
- Elle peut être utile pour initialiser certains composants, en faisant par exemple appel au modèle

```
@FXML
private void initialize() {
  // initialisation du contrôleur
}
```

Association de règles de style

```
<Label id="title" fx:id="title" text="Titre" textFill="#0022cc" ...>
```

L'attribut `id` ($\neq$ `fx:id`) définit un sélecteur CSS de type `Id` qui permet d'associer des règles de style aux composants portant cet `Id`

Exemple dans un fichier CSS `SayHello.css` :

```
#title {
  -fx-font-size: 24pt;
}
```

Dans le fichier FXML, une feuille de style (fichier CSS) peut être associé à un composant avec l'attribut `stylesheets="@CSS_File"`

```
<BorderPane stylesheets="@SayHello.css" . . . >
```

Chargement du FXML (1/2)

Cette étape peut se faire :

- soit en utilisant la méthode statique `FXMLLoader.load()`,
- soit en créant un objet de la classe `FXMLLoader`, qui permettra ensuite d'avoir accès à la racine (méthode d'instance `load()`), ainsi qu'au contrôleur si besoin :

```
public void start(Stage primaryStage) throws Exception {
  // Chargement du fichier FXML et recherche du contrôleur associé
  FXMLLoader loader =
    new FXMLLoader(getClass().getResource("SayHello.fxml"));
  BorderPane root = loader.load();
  SayHelloController ctrl = loader.getController();
  ctrl.setModel(model);
}
```

Chargement du FXML (2/2)

Dans le cas d'un contrôleur sans constructeur par défaut, on le créera et l'associera avant le chargement du fichier FXML (plus nécessaire de déclarer le contrôleur dans le FXML) :

```
public void start(Stage primaryStage) throws Exception {
  // Chargement du fichier FXML
  FXMLLoader loader =
    new FXMLLoader(getClass().getResource("SayHello.fxml"));
  // Association du contrôleur
  loader.setController(new SayHelloController("a param"));
  BorderPane root = loader.load();
}
```