Développement d'applications avec IHM (JavaFX)
Propriétés et Bindings

Petru Valicov
petru.valicov@umontpellier.fr

https://gitlabinfo.iutmontp.univ-montp2.fr/ihm

2022-2023

Contexte

Rappel - Encapsulation

En développement orienté objets 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 JavaBean

Tous les attributs privés d'une classe respectant *JavaBean* doivent être accessibles publiquement via des *getters* et modifiables grâce aux *setters*.

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

Les propriétés

Définition - Propriété

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

- Concept présent dès le début en `C#`
- "Simulé" en `Java` avec la convention *JavaBean*

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 Livre {
    private StringProperty titre = new SimpleStringProperty("titreInitial");

    public final String getTitre() {
        return titre.get();
    }

    public final StringProperty titreProperty() {
        return titre;
    }

    public final void setTitre(String titre) {
        this.titre.set(titre);
    }
}
```

À l'utilisation :

```
Livre livre = new Livre();
livre.titreProperty().set("Germinal");
System.out.println(livre.titreProperty().get());
```

Intérêt des propriétés

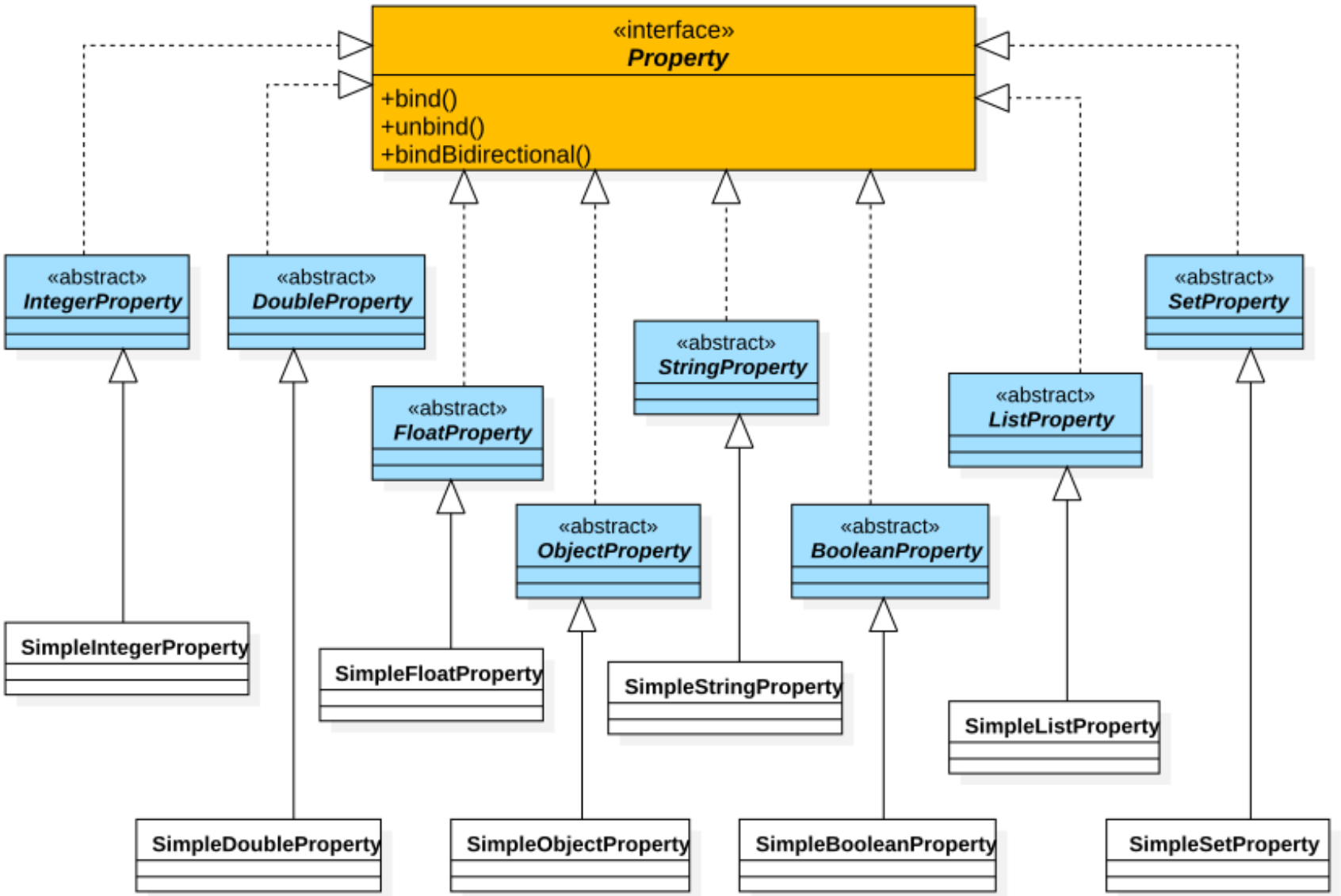
Une propriété c'est juste un objet avec des getters et setters ?

```
public interface Property<T> extends ObservableValue<T> {
    void bind(ObservableValue<? extends T> var1);

    void unbind();

    void bindBidirectional(Property<T> var1);

    void unbindBidirectional(Property<T> var1);

    void addListener(ChangeListener<? super T> listener);

    // d'autres méthodes
}
```

Les propriétés de JavaFX implémentent l'interface `Property`.

- Deux propriétés **X** et **Y** peuvent être liées (**Binding**) : le changement de **X** entraîne automatiquement la mise à jour de **Y**.
- Une propriété peut déclencher un événement lorsque sa valeur change et un écouteur (**Listener**) peut réagir en conséquence.

En `JavaFX` il existe une classe de propriété pour la plupart des types usuels, pour des collections (`List`, `Set`), pour `Object` :



Bindings

Définition - Binding

En JavaFX un *binding* est un objet qui matérialise la liaison entre une valeur donnée et une ou plusieurs valeurs observables (sources).

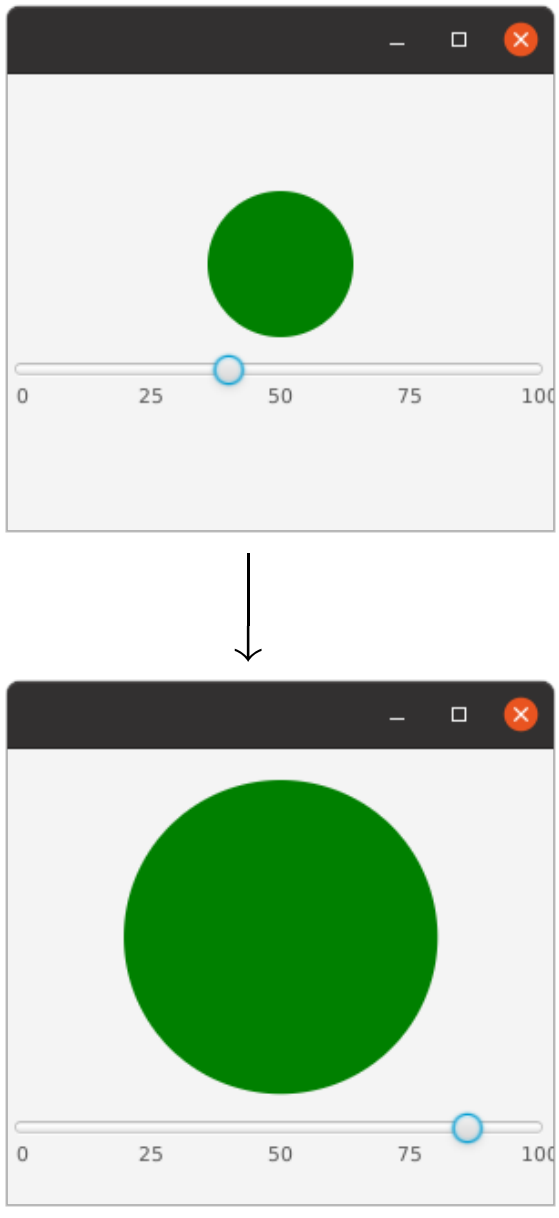
```
IntegerProperty propA = new SimpleIntegerProperty(25);
IntegerProperty propB = new SimpleIntegerProperty(41);

// création d'un binding lié à la somme des 2 propriétés propA et propB
NumberBinding somme = propA.add(propB);

System.out.println(somme.getValue()); // affiche 66

propB.setValue(100);

System.out.println(somme.getValue()); // affiche 125
```

Propriétés et Bindings

Toutes les propriétés JavaFX supportent les bindings.

Bindings - exemple

```
VBox root = new VBox();
root.setSpacing(10);
root.setAlignment(Pos.CENTER);

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

Circle cercle = new Circle();
cercle.setFill(Color.GREEN);
root.getChildren().addAll(cercle, slider);

/* binding : le rayon du cercle dépend
   de la valeur du slider */
cercle.radiusProperty().bind(slider.valueProperty());

Scene scene = new Scene(root,300,250);
primaryStage.setScene(scene);
primaryStage.show();
```



Binding **unidirectionnel** : un changement du slider entraînera un changement du rayon du cercle (mais pas l'inverse).

Bindings unidirectionnels

Dans un binding unidirectionnel, lorsqu'on lie une propriété A à une propriété B, alors changement de $B \implies$ changement de A.

```
IntegerProperty x = new SimpleIntegerProperty(10);
IntegerProperty y = new SimpleIntegerProperty(15);
IntegerProperty somme = new SimpleIntegerProperty();

// création d'un lien "addition" unidirectionnel
somme = somme.bind(x.add(y));

System.out.println(somme.getValue()); // affiche 25
x.setValue(2);
System.out.println(somme.getValue()); // affiche 17

somme.set(42); // levée d'exception RuntimeException: A bound value
               // cannot be set.
```

Une propriété ne peut être liée de manière unidirectionnelle qu'à une seule autre propriété : un seul `bind()` possible.

Bindings bidirectionnels

Dans un binding bidirectionnel, lorsqu'on lie une propriété A à une propriété B, les changements sont réciproques

```
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` possède des méthodes statiques utiles pour créer des liaisons **haut niveau** :

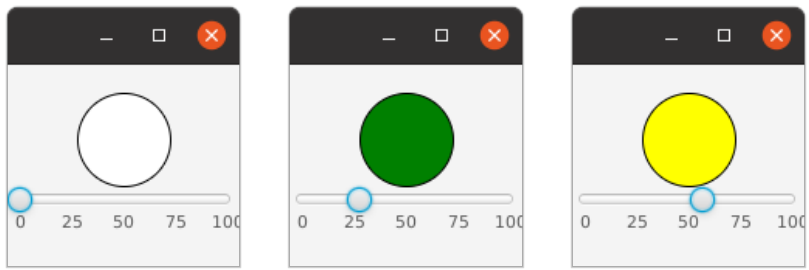
- calculs : `add`, `divide`, `multiply`, `max`, `min`, etc.
- logique : `and`, `or`, `equal`, `lessThan`, `greaterThan`, etc.
- conversion en chaînes de caractères : `concat`, `convert`, etc.
- expression ternaire : `when(condition).then(value1).otherwise(value2)`

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

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

Création des bindings haut niveau

```
import static javafx.beans.binding.Bindings.when;
...

    public void start(Stage primaryStage) {
        VBox root = new VBox();
        Slider slider = new Slider();
        slider.setShowTickLabels(true);

        Circle cercle = new Circle(50);
        cercle.setStroke(Color.BLACK);
        root.getChildren().addAll(cercle, slider);
        root.setAlignment(Pos.CENTER);

        cercle.fillProperty().bind(
                when(slider.valueProperty().lessThan(20))
                        .then(Color.WHITE)
                        .otherwise(when(slider.valueProperty().lessThan(40))
                                .then(Color.GREEN)
                                .otherwise(when(slider.valueProperty().lessThan(60))
                                        .then(Color.YELLOW)
                                        .otherwise(when(slider.valueProperty().lessThan(80))
                                                .then(Color.RED)
                                                .otherwise(Color.BLUE)))));

        Scene scene = new Scene(root, 200, 170);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```



Bindings bas niveau

- Parfois les bindings haut niveau ne sont pas suffisant pour exprimer les dépendances complexes.
- Dans ce cas il est possible de définir une implémentation d'une des classes abstraites de binding :

```
Slider montantEmprunt = new Slider((500, 10000, 100);
Slider interets = new Slider(0.5, 5, 0.7);
Slider duree = new Slider(1, 20, 5);
Label mensualité = new Label("0");

// création d'un DoubleBinding bas niveau (low-level) avec une classe anonyme
DoubleBinding binding = new DoubleBinding() {
    // constructeur de la classe anonyme (qui hérite de DoubleBinding)
    {
        // appel de la méthode bind de la classe-mère (DoubleBinding)
        super.bind(montantEmprunt.valueProperty(), interets.valueProperty(),
              duree.valueProperty());
    }

    @Override
    protected double computeValue() {
        double numerateur = montantEmprunt.getValue() * interets.getValue();
        double denominateur = (1 - 1 / Math.pow(1 + interets.getValue(), duree.getValue() * 12));
        return numerateur / denominateur;
    }
};

mensualité.textProperty().bind(Bindings.convert(binding));
```

Observation des propriétés

- L'interface `Property` implémente l'interface `Observable`
- Il est donc possible d'ajouter des écouteurs sur une propriété

```
public void start(Stage primaryStage) {

    Circle cercle = new Circle(20);

    BorderPane root = new BorderPane();
    root.setCenter(cercle);
    Scene scene = new Scene(root, 120, 50);

    // ajout d'un écouter qui provoque le changement du rayon du cercle
    scene.widthProperty().addListener(
            (source, ancienneValeur, nouvelleValeur) ->
                 cercle.setRadius(nouvelleValeur.doubleValue()/4));

    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Observation des propriétés

- L'interface `Property` implémente l'interface `Observable`
- Il est donc possible d'ajouter des écouteurs sur une propriété

```
public void start(Stage primaryStage) {
    Circle cercle = new Circle(20);
    BorderPane root = new BorderPane();
    root.setCenter(cercle);
    Scene scene = new Scene(root, 120, 50);

    // ajout d'un écouter avec une classe anonyme
    scene.widthProperty().addListener(
            new ChangeListener<Number>() {
                @Override
                public void changed(ObservableValue<? extends Number> source, Number
                        ancienneValeur, Number nouvelleValeur) {
                    cercle.setRadius(nouvelleValeur.doubleValue() / 4);
                }
            }
    );

    primaryStage.setScene(scene);
    primaryStage.show();
}
```