JavaFX Properties and Binding

Josh Wilcox (jw14g24)

April 29, 2025

Table of Contents

JavaFX Property Classes

What are Properties

- Properties are JavaFX **wrappers** for Java Primitives and Objects
- They afford a mechanism called **Binding** that allows you to keep two components synchronised
  - When one object is changed - it can ensure that all other objects that want to be affected by that change are indeed changed

JavaFX Specifics

- Take for example the `TextField` and `Label` classes
  - Both of these classed inherit from the `Labeled` class
  - They both have an attribute called `text` of type `StringProperty` which is access through `textProperty()`
- Also take the `Button` class
  - We also have `BooleanProperty` properties in buttons
  - `cancelButton` - Receives a keyboard ESC press
  - `defaultButton` - Receives a keyboard ENTER press

Object Property Classes

- `StringProperty` - wraps `String`
- `ObjectProperty` - wraps any object of type `T`
- `ListProperty` - wraps an `ObservableList`
- `MapProperty` - wraps `ObservableMap`
- `SetProperty` - wraps `ObservableSet`

JavaFX Binding

Overview

- **Binding** allows to keep two components synchronised
- Often in GUI, we want to pair interface elements with the underlying data model
- Previously, we would change UI elements directly through event handlers - however this can quickly become messy
  - Binding is a much tidier alternative
- JavaFX uses Property objects to implement binding
  - These objects automatically notify listeners when their values are changed

The Hollywood Principle

- A `Don't Call us, We'll Call you` principle
  - Rather than objects keeping on asking for updates, the ”Boss” object will let the lower objects know through an announcement
- Example: The label of a JavaFX object will always scan the value of a binded variable and will set it’s label to that variable

Java Example of Binding

```java
public class SimpleBinding extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Create a shared string property for binding
        StringProperty sharedText = new SimpleStringProperty("Don't Panic");

        // Create UI controls
        Button button = new Button("Press me in case of fire");
        Label label1 = new Label();
        Label label2 = new Label();
        Label label3 = new Label();

        // Bind all labels to the shared text property
        label1.textProperty().bind(sharedText);
        label2.textProperty().bind(sharedText);
        label3.textProperty().bind(sharedText);

        // Set button action to update shared text
        button.setOnAction(event -> sharedText.set("FIRE: Run away"));

        // Create layout and scene
        VBox root = new VBox(10); // 10 pixels spacing
        root.getChildren().addAll(button, label1, label2, label3);
        Scene scene = new Scene(root, 300, 200);

        // Set up and show the stage
        primaryStage.setTitle("Simple Binding Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

ChangeListener

Overview

- The `ChangeListener` object is a **functional interface** for responding automatically when the value of an **observable** property changed
- It contains a single method: `changed(ObservableValue observable, T oldValue, T newValue)`
  - This method is called whenever the observed value changes
  - The parameters provide access to what changed and its old/new values

Implementation

- Can be implemented using lambda expressions
- Common use cases:
  - Monitoring text field input changes
  - Tracking slider value modifications
  - Observing selection changes in lists
- Added to properties using `addListener()` method

Example of using a ChangeListener

```
1  // SimpleIntegerProperty is a JavaFX class that wraps an integer value and allows
2  // for observation of value changes
3  SimpleIntegerProperty counter = new SimpleIntegerProperty(0);
4
5  // First approach: Using an anonymous class to implement ChangeListener
6  // ChangeListener<T> is a functional interface that handles value changes
7  counter.addListener(new ChangeListener<Number>() {
8      @Override
9      public void changed(ObservableValue<? extends Number> observable,
10                         Number oldValue, Number newValue) {
11         // This method is called whenever the counter value changes
12         // It receives the observable property, old value, and new value
13         System.out.println("Counter changed from " + oldValue + " to " + newValue);
14     }
15 });
16
17 // Second approach: Using a lambda expression (more concise modern syntax)
18 // Lambda expressions can replace anonymous classes for functional interfaces
19 counter.addListener((observable, oldValue, newValue) ->
20     System.out.println("A lambda version " + oldValue + " to " + newValue)
21 )
```