Java FX binding

Pieter van den Hombergh
Thijs Dorssers
Richard van den Ham

Fontys Hogeschool voor Techniek en Logistiek

May 12, 2018

Contents of this talk

JavaFX Properties and Binding

FX Beans vs Java Beans

- more direct binding/ relationship of and between variables.
  - The work around triggering and binding does no longer take place in containing class.
  - because the properties are “exposed”, so can manipulated from the outside.
- new (extra) convention for getter of property. `xXXXProperty()` where xXXX is the name of the property.

Meet Bill

Simple bill example

```
public class Bill {
// Define a variable to store the property

    private final DoubleProperty amountDue = new SimpleDoubleProperty↩
        ();

    // Define a getter for the property's value
    public final double getAmountDue() {
        return amountDue.get();
    }

    // Define a setter for the property's value
    public final void setAmountDue( double value ) {
        amountDue.set( value );
    }

    // Define a getter for the property itself
    public final DoubleProperty amountDueProperty() {
        return amountDue;
    }
}
```

Improvements

- JavaFX properties expands and improves the JavaBeans model.
- **Binding** is assembled from one or more sources, the *dependencies*.
- A binding observes its dependencies for changes and updates itself after change detection, but lazily.
- Lazy means, that the value is recomputed on demand (when getter is called) instead of every time it is set.
- The documentation also states that the implementer should minimize the work in event handlers: *"Implementations of this class should strive to generate as few events as possible to avoid wasting too much time in event handlers."*
- So:
  - if the new value is the same as the previous: do not bother.
  - If you are already invalid, do not propagate this news any further.

Property APIs

JavaFX beans provide three way to deal with bound properties:

- High level API.
  - Fluent API
  - Bindings utility class.
- Low level API.

High level API, Fluent

Make use of the (computational) method in the property: `NumberProperty` and its children.

Fluent API

```
public static void main( String[] args ) {
    IntegerProperty num1 = new SimpleIntegerProperty( 1 );
    IntegerProperty num2 = new SimpleIntegerProperty( 2 );
    IntegerProperty num3 = new SimpleIntegerProperty( 5 );
    NumberBinding sumMul = num1.add( num2 ).multiply( num3 );
    System.out.println( sumMul.getValue() );
    num1.set( 2 );
    System.err.println( sumMul.getValue() );
}
```

Note the use of *chaining* the method calls. It is because the `add` etc. methods return an object which support the operation.
This is called a *fluent* style of programming.

High level API, Bindings class

The same computation can be done using the `Bindings` class which provides *factories* for all kinds of binding.

Using Bindings

```
public static void main( String[] args ) {
    IntegerProperty num1 = new SimpleIntegerProperty( 1 );
    IntegerProperty num2 = new SimpleIntegerProperty( 2 );
    IntegerProperty num3 = new SimpleIntegerProperty( 5 );
    // note the use of static import
    NumberBinding sum = multiply( num3, add( num1, num2 ) );
    System.out.println( sum.getValue() );
    num1.setValue( 2 );
    System.err.println( sum.getValue() );
}
```

The `add` and `multiply` methods stem from the Bindings class.

The import statement

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

modified from javafx bindings tutorial.

Combining Fluent API and Bindings

```
public static void main( String[] args ) {
    IntegerProperty num1 = new SimpleIntegerProperty( 1 );
    IntegerProperty num2 = new SimpleIntegerProperty( 2 );
    IntegerProperty num3 = new SimpleIntegerProperty( 3 );
    IntegerProperty num4 = new SimpleIntegerProperty( 4 );
    NumberBinding total = Bindings.add( num1.multiply( num2 ),
                                        num3.multiply( num4 ) );
    System.out.println( total.getValue() );
    num1.setValue( 2 );
    System.err.println( total.getValue() );
}
```

- Convenient to transform a formula to binding.
- When mixing types (e.g. int, double), conversion (promotion) can take place. The rules are the same as in Java.

1. If one of the operands is a double, the result is a double.
2. If not and one of the operands is a float, the result is a float.
3. If not and one of the operands is a long, the result is a long.
4. The result is an integer otherwise.

from javafx bindings tutorial.

Lazy evaluation

Lazy evaluation in action

```
NumberBinding total = Bindings.add(
        bill1.amountDueProperty()
        .add( bill2.amountDueProperty() ),
        bill3.amountDueProperty() );

// First call makes the binding invalid
bill1.setAmountDue( 200.00 );

// The binding is now invalid
bill2.setAmountDue( 100.00 );
bill3.setAmountDue( 75.00 );

// Make the binding valid...
System.out.println( total.getValue() );

// Make invalid...
bill3.setAmountDue( 150.00 );

// Make valid...
System.out.println( total.getValue() );
```

The title is probably an oxymoron.

Low Level API

```
final DoubleProperty a = new SimpleDoubleProperty( 1 );
final DoubleProperty b = new SimpleDoubleProperty( 2 );
final DoubleProperty c = new SimpleDoubleProperty( 3 );
final DoubleProperty d = new SimpleDoubleProperty( 4 );

DoubleBinding db = new DoubleBinding() {

    {
        super.bind( a, b, c, d );
    }

    @Override
    protected double computeValue() {
        return ( a.get() * b.get() ) + ( c.get() * d.get() );
    }
};

System.out.println( db.get() );
b.set( 3 );
System.err.println( db.get() );
```

- The low level API can give more control and performance (avoids intermediate objects) in return for a little more work
- Note the use of the block, which effectively is the sole constructor.
- `super` of DoubleBinding takes care of invalidation behavior.

Any open issues?

Not all understood?

- see `https://docs.oracle.com/javase/8/javafx/properties-binding-tutorial/binding.htm`

Questions?

Questions or remarks?