# JavaFX

Using JavaFX Properties and Binding

Release 2.1

**E20469-04**

June 2013

ORACLE®

JavaFX/Using JavaFX Properties and Binding, Release 2.1

E20469-04

Primary Author: Scott Hommel

Contributing Author:

Contributor:

# Contents

1

# Using JavaFX Properties and Binding

In this tutorial you learn how to use properties and binding in JavaFX 2 applications.

The tutorial describes relevant APIs and provides working examples that you can compile and run.

## Overview

For many years, the Java programming language has used the JavaBeans component architecture to represent the property of an object. This model consists of both an API and a design pattern; it is widely understood by Java application developers and development tools alike. This release introduces property support into JavaFX, support that is based on the proven JavaBeans model, but expanded and improved.

JavaFX properties are often used in conjunction with binding, a powerful mechanism for expressing direct relationships between variables. When objects participate in bindings, changes made to one object will automatically be reflected in another object. This can be useful in a variety of applications. For example, binding could be used in a bill invoice tracking program, where the total of all bills would automatically be updated whenever an individual bill is changed. Or, binding could be used in a graphical user interface (GUI) that automatically keeps its display synchronized with the application's underlying data.

Bindings are assembled from one or more sources, known as *dependencies*. A binding observes its list of dependencies for changes, and then updates itself automatically after a change has been detected.

The binding APIs are divided into two broad categories:

1. The High-Level API: Provides a simple way to create bindings for the most common use cases. Its syntax is easy to learn and use, especially in environments that provide code completion, such as the NetBeans IDE.
2. The Low-Level API: Provides additional flexibility, and can be used by advanced developers in situations where the High-Level API is insufficient. The Low-Level API was designed for fast execution and small memory footprint.

The remainder of this tutorial describes these APIs, and provides working code examples that you can compile and run.

## Understanding Properties

As mentioned in the overview, JavaFX property support is based on the well-known property model established by the JavaBeans component architecture. This section provides a brief overview of what that means, then explains how properties apply to JavaFX.

The Java programming language makes it possible to encapsulate data within an object, but it does not enforce any specific naming conventions for the methods that you define. For example, your code might define a `Person` class, which encapsulates a first name and a last name. But without naming conventions, different programmers might choose different names for these methods: `read_first()`, `firstName()`, `getFN()`, etc. would all be perfectly valid choices. However, there is no guarantee that these names will be meaningful to other developers.

The JavaBeans component architecture addressed this problem by defining some simple naming conventions that bring consistency across projects. In JavaBeans programming, the full signatures for these methods would be: `public void setFirstName(String name)`, `public String getFirstName()`, `public void setLastName(String name)`, and `public String getLastName()`. This naming pattern is easily recognizable, both to human programmers and to editing tools, such as the NetBeans IDE. In JavaBeans terminology, the `Person` object is said to contain `firstName` and `lastName` *properties*.

The JavaBeans model also provides support for complex property types, plus an event delivery system. It also contains a number of support classes, all available as an API under the `java.beans` package. Therefore, mastering JavaBeans programming involves learning the required naming conventions and its corresponding API. (For more background reading on JavaBeans in general, see the JavaBeans lesson of the Java Tutorial at `http://download.oracle.com/javase/tutorial/javabeans`).

Similarly, understanding JavaFX properties also requires learning a few new APIs and naming conventions. In JavaFX, it is entirely possible that you will only be interested in *using* classes that contain properties (as opposed to implementing properties in your own custom classes), but Example 1–1 will familiarize you with the new method naming conventions that form the JavaFX property pattern. It defines a class named `Bill`, which implements a single property named `amountDue`.

***Example 1–1 Defining a Property***

```
package propertydemo;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

class Bill {

    // Define a variable to store the property
    private 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 DoubleProperty amountDueProperty() {return amountDue;}

}
```

The `amountDue` object — an instance of the `javafx.beans.property.DoubleProperty` class — is marked as `private` to encapsulate it from the outside world. This is standard practice in both Java and JavaBeans application development. Note however that the object's type is not one of the standard Java primitives, but rather, a new wrapper class that encapsulates a Java primitive and adds some extra functionality (the classes under `javafx.beans.property` all contain built-in support for observability and binding as part of their design).

The property method naming conventions are as follows:

- The `getAmountDue()` method is a standard getter that returns the current value of the `amountDue` property. By convention, this method is declared as `final`. Note that the return type for this method is `double,` not `DoubleProperty`.
- The `setAmountDue(double)` method (also `final`) is a standard setter that allows a caller to set the property's value. The setter method is optional. Its parameter is also of type `double`.
- Finally, the `amountDueProperty()` method defines the property getter. This is a new convention in which the method name contains the name of the property (`amountDue`, in this case), followed by the word "Property." The return type is the same as the property itself (`DoubleProperty`, in this example).

When building GUI applications with JavaFX, you will notice that certain classes in the API already implement properties. For example, the `javafx.scene.shape.Rectangle` class contains properties for `arcHeight`, `arcWidth`, `height`, `width`, `x`, and `y`. For each of these properties there will be corresponding methods that match the conventions previously described. For example, `getArcHeight()`, `setArcHeight(double)`, `arcHeightProperty()`, which together indicate (to both developers and tools) that the given property exists.

You can also add a change listener to be notified when the property's value has changed, as shown in Example 1–2.

***Example 1–2 Using a ChangeListener***

```
package propertydemo;

import javafx.beans.value.ObservableValue;
import javafx.beans.value.ChangeListener;

public class Main {

    public static void main(String[] args) {

      Bill electricBill = new Bill();

       electricBill.amountDueProperty().addListener(new ChangeListener(){
        @Override public void changed(ObservableValue o,Object oldVal,
                 Object newVal){
             System.out.println("Electric bill has changed!");
        }
      });

      electricBill.setAmountDue(100.00);

    }
}
```

Running this example will print the message "Electric bill has changed" to standard output, proving that the change listener notification is working.

## Using the High-Level Binding API

The High-Level API is the quickest and easiest way to begin using bindings in your own applications. It consists of two parts: the Fluent API, and the `Bindings` class. The Fluent API exposes methods on the various dependency objects, whereas the `Bindings` class provides static factory methods instead.

To begin using the Fluent API, consider a simple use case in which two integers are bound so that their values are always added together. In Example 1–3, there are three variables involved: `num1` (a dependency), `num2` (a dependency), and `sum` (the binding). The dependency types are both `IntegerProperty`, and the binding itself is `NumberBinding`.

***Example 1–3 Using the Fluent API***

```
package bindingdemo;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.binding.NumberBinding;

public class Main {

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

This code binds the two dependencies, prints their sum, then changes the value of `num1` and prints the sum again. The results are "3" and "4", which proves that the binding is working.

You could also use the `Bindings` class to do the same thing, as shown in Example 1–4.

***Example 1–4 Using the Bindings Class***

```
package bindingdemo;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.binding.NumberBinding;
import javafx.beans.binding.Bindings;

public class Main {

    public static void main(String[] args) {
       IntegerProperty num1 = new SimpleIntegerProperty(1);
       IntegerProperty num2 = new SimpleIntegerProperty(2);
       NumberBinding sum = Bindings.add(num1,num2);
       System.out.println(sum.getValue());
```

```
        num1.setValue(2);
        System.err.println(sum.getValue());
    }
}
```

Example 1–5 combines the two approaches:

***Example 1–5 Combining Both Approaches***

```
package bindingdemo;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.binding.NumberBinding;
import javafx.beans.binding.Bindings;

public class Main {

    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());
    }
}
```

Example 1–5 modifies the code to invoke the `multiply` method from the Fluent API, and `add` from the `Bindings` class. You should also know that the High-Level API lets you mix types when defining arithmetic operations. The type of the result is defined by the same rules as the Java programming language:

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.

The next section explores observability, and demonstrates how invalidation listeners differ from change listeners.

## Exploring Observable, ObservableValue, InvalidationListener, and ChangeListener

The binding API defines a set of interfaces that enable objects to be notified when a value change or invalidation takes place. The `Observable` and `ObservableValue` interfaces fire the change notifications, and the `InvalidationListener` and `ChangeListener` interfaces receive them. The difference between the two is that `ObservableValue` wraps a value and fires its changes to any registered `ChangeListener`, whereas `Observable` (which does *not* wrap a value) fires its changes to any registered `InvalidationListener`.

The JavaFX binding and property implementations all support lazy evaluation, which means that when a change occurs, the value is not immediately recomputed. Recomputation happens later, if and when the value is subsequently requested.

In Example 1–6, the bill total (a binding) will be marked as invalid the first time it detects a change in one of its dependencies. However, the binding object will recalculate itself only if the total is actually requested again.

***Example 1–6 Using an InvalidationListener***

```
package bindingdemo;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.binding.NumberBinding;
import javafx.beans.binding.Bindings;
import javafx.beans.InvalidationListener;
import javafx.beans.Observable;

class Bill {

    // Define the property
    private 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 DoubleProperty amountDueProperty() {return amountDue;}

}

public class Main {

    public static void main(String[] args) {

        Bill bill1 = new Bill();
        Bill bill2 = new Bill();
        Bill bill3 = new Bill();

        NumberBinding total =
          Bindings.add(bill1.amountDueProperty().add(bill2.amountDueProperty()),
              bill3.amountDueProperty());
        total.addListener(new InvalidationListener() {

        @Override public void invalidated(Observable o) {
                System.out.println("The binding is now invalid.");
            }
        });

        // 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());
    }
}
```

By changing the value of a single bill, the binding becomes invalid, and the invalidation listener will fire. But if the binding is already invalid, the invalidation listener will not fire again, even if another bill changes. (In Example 1–6, invoking `total.getValue()` moves the binding from invalid to valid.) We know this because a subsequent change to any bill in the dependency list will cause the invalidation listener to fire again. This would not happen if the binding was still invalid.

Note that registering a `ChangeListener` will enforce eager computation, even if the implementation of the `ObservableValue` supports lazy evaluation. For a lazily evaluated value, it is not possible to know if an invalid value really has changed until it is recomputed. For this reason, generating change events requires eager evaluation, while invalidation events can be generated for both eager and lazy implementations.

## Using the Low-Level Binding API

If the High-Level API is not enough to satisfy your requirements, you can always use the Low-Level API instead. The Low-Level API is for developers who require more flexibility (or better performance) than that offered by the High-Level API.

Example 1–7 shows a basic example of using the Low-Level API.

***Example 1–7 Using the Low-Level API***

```
package bindingdemo;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.binding.DoubleBinding;

public class Main {

    public static void main(String[] args) {

        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.out.println(db.get());
        }
    }
```

Using the Low-Level API involves extending one of the binding classes and overriding its `computeValue()` method to return the current value of the binding. Example 1–7 does this with a custom subclass of `DoubleBinding`. The invocation of `super.bind()` passes the dependencies up to `DoubleBinding` so that the default invalidation behavior is retained. It is generally not necessary for you to check if the binding is invalid; this behavior is provided for you by the base class.

You now know enough information to begin using the Low-Level API.