JavaFX Property Architecture

For many years, Java has used the JavaBeans API to represent a *property* of an object. This pattern is generally very highly regarded in the Java world. Other modern languages (such as C#) have explicit language support for creating properties, whereas JavaBeans is both an API and design pattern that you must apply to your objects. While porting from JavaFX Script to Java, it was important that we settle on the pattern to use for representing properties. Clearly, leveraging the JavaBeans pattern has its advantages, such as the ability to work with additional tools and languages from the start (which already know how to deal with JavaBeans) and familiarity with Java programmers. However there were some aspects to the JavaBeans pattern and API which were less desirable, such as the reliance on String representations of properties (to reference which property had changed, for example), or the event model which forced boxing and eager evaluation. Clearly, some adaptation of JavaBeans had to be done for JavaFX.

A property generally has the following characteristics:

- It is encapsulated
    - The property might be computed on demand
    - The property storage (the field that backs it) might change representations over time
    - The class defining the property can intercept mutations (calls to the setter) in order to maintain invariants
- It can be observable
- It can be read-only
- Subclasses can react to changes in the property
- Properties have names
- Properties have types
- Properties may have meta data

JavaFX expands this list of capabilities by also allowing any writable property to be bound, or unbound. The primary complication we face in JavaFX is support for lazy binding (and to a lesser extent, observability) in an efficient manner. The secondary complication that we face is how to make the code simple to read and write, such that 3rd party developers can also write beans with properties.

In years past there have been raging debates regarding properties in Java. During those debates several alternative designs to JavaBeans were proposed and discussed. Each of these alternative designs were also considered for JavaFX, yet in the end, none of them fit our needs quite right. During the port from JavaFX Script to Java, we initially went with a design which had good performance characteristics (though, as we will see later, perhaps not ideal), but at the cost of massive boilerplate per property. Due to this, and to the fact that we had not yet solidified the design, we used Lombok as a tool to help us generate the boilerplate. This proved to be very instrumental in allowing the teams to port to Java while the "foundation" moved around. Yet Lombok also proved to be a pain for the IDEs to understand and utilize, for documentation, for readability, and for debugging and performance testing. This document represents the current design, which reduces boilerplate and removes the reliance on Lombok.

Property Solutions

It is easy to get confused regarding the different approaches to properties, but understanding those different approaches is important in order to understand why we ended up at our final design. The terminology for these approaches are gathered from Stephen Colebourne's blog on Java 7 Properties Terminology. If you haven't read it yet, go read it now. Done? Good.

The **Bean-independent** property pattern is a very lightweight pattern in terms of per-bean and per-property overhead. There is virtually none. It works reasonably well for frameworks, though is more cumbersome to code against as you must have both a reference to the bean, and the Bean-independent property (similar to the java.lang.reflect.Method class or other reflection classes). Generally, bean-independent properties are not intended to be used by developers under normal circumstances.

The **Bean-attached** property pattern is much heavier than the bean-independant one, because it requires an object-per-property-per-instance rather than simply an object-per-property-per-class. Like bean-independent properties, bean-attached properties are very handy for frameworks such as binding and animations. In the bean-attached property pattern, the "property" object is simply a wrapper which invokes the getter and setter. Typically instances of bean-attached properties can be created on the fly rather than actually cached and reused. Although it is "heavier" than Bean-independent properties, it is also easier to use and more natural for developers.

Both bean-independent and bean-attached properties are adjuncts to the existing JavaBeans pattern. There is still a getter and setter and a field for the variable. Observability is not handled out-of-the-box, but is the responsibility of the setter to implement, ostensibly by continuing to use the JavaBeans event API.

The **Stateful** property pattern is the heaviest of all. It requires each property to be defined in terms of a Property object. This object holds the field, implements observability, and handles all other aspects of the property. Indeed, it is defined as public **final** Property name = new Property(). Note that it is a public final field. The code for interacting with a property object is incredibly nice (one might say, silky smooth). Unfortunately it suffers from several fatal flaws, including the lack of encapsulation (the bean defining the property cannot hide its definition), and most critically, the inability to expose a different API to external consumers vs. package, protected, or private consumers. For example, if you wanted to allow external classes to only be able to read a property, but allow the class itself to write to the property, you are out of luck (without severe gymnastics).

Initially in the Java port of JavaFX, we used a new pattern which I will call **Enhanced** properties. This pattern starts with the standard JavaBeans pattern, adds a few more methods (store, onInvalidate, model), and adds a mixture of Bean-attached and Bean-independent properties. If a framework wants to get a Bean-independent representation of the property, they can simply take the PropertyReference (which is a public static final field created for each property). There are some use cases (such as the comparator example Stephen gave) which are served very well by such a field. In addition, it allowed us to have a type-safe reference to the property (whereas traditional JavaBeans required a String). If you wanted a Bean-attached representation of the property, you can simply call the "model" method to get a reference to the PropertyModel which can then be passed to a framework (such as animation and binding).

The "store" method is a convenience method which gets called at a certain point during the execution of the setter. The fundamental advantage of the store method is that it allows the subclass to insert some logic that occurs either just before the field is updated, or just after the field is updated but **before** events are fired. Otherwise, the "store" method is not a critical part of the design.

Finally, due to the addition of binding, it is important that we have the onInvalidated method. It is called either when the setter for a property is called, or when the property is bound and the binding becomes invalid. This is critical to implement proper lazy binding.

The Enhanced property pattern is very heavy on boilerplate code, which is its primary drawback. A secondary drawback is the somewhat convoluted event system which requires a bean to implement the Bean interface if it is to be observable. The primary benefits of the Enhanced property pattern are power and flexibility, and support for lazy binding.

JavaFX Solution

The JavaFX solution is a combination of the Bean-attached and Stateful property patterns. It does away with the Bean interface and has each property itself implement observability. While at first glance it appears to balloon the memory requirements for a property, in actuality it may end up being *more efficient* than the Enhanced property pattern (and certainly more efficient than the Stateful property pattern). Both lazy binding and observability are built into the design directly. Because the JavaFX Property pattern uses getters and setters and hides the actual field definition of the property, many forms of optimization are possible.

JavaFX Properties are comprised of three methods (as opposed to the two methods which make up traditional JavaBean properties). They are:

- final getter
- final setter (optional)
- property getter

Here is an example of a very trivial property implementation:

Person.java

```
public class Person {
    private StringProperty name = new StringProperty("Unnamed");
    public final String getName() { return name.get(); }
    public final void setName(String value) { name.set(value); }
    public StringProperty nameProperty() { return name; }
}
```

As you can see from this example, a trivial property implementation is very clean and to the point. The getter and setter are simply convenience methods (with very important utility in writing performance critical property implementations, as you will see later!). They delegate directly to the property itself.

The property getter is named fooProperty, where "foo" is the name of the property. This method is not final, and simply returns a reference to the property.

The key feature of this design is that the property object implements all of the logic regarding adding and removing listeners, and binding. A consumer of this API can get a reference to the property and invoke the **get** and **set** methods directly on it, they don't have to use the getter or setter.

This article began with a series of criteria which the properties pattern had to meet for JavaFX. We will go through the entire list, demonstrating the manner in which the JavaFX Property pattern satisfies the design constraints. This design is very flexible in what it allows in the implementation. There are more ways to implement the pattern than what is shown in this document. At the end of this document is a "cookbook" listing different ways to implement properties, and listing the various pros and cons to each choice (usually some trade-off among static footprint, dynamic footprint, and runtime performance).

Properties Must Be Observable

The JavaFX Property pattern perfectly satisfies this constraint. All property objects are inherently observable, even read-only properties (as opposed to immutable properties, which can also be observed although it is pointless to do so). Every Property object has built into it all the machinery to support observability. The client of this API simply has to add a change listener to the property object, like this:

```
rectangle.xProperty().addListener(new ChangeListener() {
    @Override public void onChange(Property p) {
        // do something
    }
});
```

Properties Can Be Read-Only

A property can be an instance of Property or ReadOnlyProperty. While the implementation can take pains to ensure that the ReadOnlyProperty returned from the property getter is actually a different object from the property object used for storage, this is an implementation decision and not required by the pattern. Property extends from ReadOnlyProperty.

ReadOnlyNamePerson.java

```
public class ReadOnlyNamePerson {
    private StringProperty name = new StringProperty("Unnamed");
    public final String getName() { return name.get(); }
    private final void setName(String value) { name.set(value); }
    public ReadOnlyStringProperty nameProperty() { return name; }
}
```

As demonstrated in the above, there are two things that must happen to make a property read only. First, a ReadOnlyProperty subtype must be returned from the property getter. Second, the setter needs to be modified to support the proper access.

The setter could be one of "protected", "private", default (package), or can be removed entirely. The property getter returns a ReadOnlyProperty in all of these cases, because to the public world, the property is read only, even if the class or one of its subclasses or package classes can mutate the property.

Properties Can Have Custom Setters

Because the property can be set either through the setter, or directly through the property object itself, custom setter logic must not be placed in the setter, but rather in the property object itself.

NullNamesNotAllowedPerson.java

```
public class NullNamesNotAllowedPerson {
    private StringProperty name = new StringProperty("Unnamed") {
        @Override public void set(String value) {
            if (value == null) throw new NullPointerException("Null names not allowed");
            super.set(value);
        }
    };
    public final String getName() { return name.get(); }
    public final void setName(String value) { name.set(value); }
    public StringProperty nameProperty() { return name; }
}
```

You can provide a unique subclass of the property that enforces the constraints you wish, or that reacts to changes. Do note however that the set method is not called when the property is bound, so if you need to perform some kind of logic both when the property is bound and when it is set, then you probably should override the "invalidate" method instead of the "set" method.

Subclasses Must Be Able To React To Changes In The Superclass Property

There are two ways in which JavaFX Properties can be made to support this requirement. The first is to let the subclass override the property getter, supplying its own implementation for the property which wraps the one supplied by the super class. This approach has some cost in terms of static and dynamic footprint.

SubclassForcesUppercaseNames.java

```
public class SubclassForcesUppercaseNames extends Person {
    private StringProperty name;
    @Override public StringProperty nameProperty() {
        if (name == null) {
            final StringProperty superName = super.nameProperty();
            name = new StringPropertyDelegate(super.nameProperty()) {
                @Override public void set(String value) {
                    super.set(value.toUpperCase());
                }
            };
        }
        return name;
    }
```

Here I assume there is some Property implementation which is based on delegation, such that every call to get, set, add / remove listener, or bind / unbind flows through to the super property. It then overrides the set to do some special logic before delegating to the superName property.

Another approach is for the superclass to provide an explicit protected method which the subclass is intended to implement. This is useful in cases where high-performance is required (either in terms of reduced static or dynamic footprint). In a search through our code base, it is not clear that this is really needed much if at all. There are only some 20+ cases where we override a property in a subclass, and the delegation solution is probably sufficient for those cases.

SubclassForcesUppercaseNames2.java

```
public class Person {
    private StringProperty name = new StringProperty("Unnamed") {
        @Override public void set(String value) {
            super.set(modifySetName(value));
        }
    }
    public final String getName() { return name.get(); }
    public final void setName(String value) { name.set(value); }
    public StringProperty nameProperty() { return name; }
    protected String modifySetName(String value) { return value; }
}

public class SubclassForcesUppercaseNames2 extends Person {
    @Override protected String modifySetName(String value) {
        return value.toUpperCase();
    }
}
```

Properties Must Support Encapsulation

The JavaFX Property pattern supports encapsulation. There are no public fields of any kind. As such, when and how a property object is created is left to the implementation. Properties can be created lazily (using several different approaches). Any property subclass can be created, including anonymous subclasses, rather than using the built-in standard properties.

Properties Must Have Names and Types

The properties in this pattern have both names and types. There are Property types for each of the primitives, String, and then any Object<T>.

Properties May Have Metadata

Any meta-data for a property is provided as annotations on the property getter.

The Property Might Be Computed On Demand

Computed properties are heavier in JavaFX than they were in Java, primarily because all properties whether or not computed must support change events and listener lists. As such, we use binding to implement computed properties.

ComputedProperty.java

```
public class Rectangle {
    private DoubleProperty x = new DoubleProperty();
    private DoubleProperty width = new DoubleProperty();

    private DoubleProperty x2 = new DoubleProperty(); {
        x2.bind(x.add(width));
    }
    public ReadOnlyDoubleProperty x2Property() { return x2; }
}
```

More Examples

The following is a somewhat larger example showing not only how to define a property, but also how to use one. Notice that using a property is very simple and straightforward. You can either use the getter and setter directly, or get the property in order to add a listener or bind to it.

Rectangle.java

```
public class Rectangle extends Shape {
    private DoubleProperty x = new DoubleProperty();
    public DoubleProperty xProperty() { return x; }
    public double getX() { return x.get(); }
    public void setX(double value) { x.set(value); }

    private DoubleProperty y = new DoubleProperty();
    public DoubleProperty yProperty() { return y; }
    public double getY() { return y.get(); }
    public void setY(double value) { y.set(value); }

    private DoubleProperty width = new DoubleProperty();
    public DoubleProperty xProperty() { return x; }
    public double getX() { return x.get(); }
    public void setX(double value) { x.set(value); }

    private DoubleProperty height = new DoubleProperty();
    public DoubleProperty xProperty() { return x; }
    public double getX() { return x.get(); }
    public void setX(double value) { x.set(value); }
}
```

MyApp.java

```
public class MyApp extends Application {
    @Override public void start(Stage stage) {
        final Rectangle r1 = new Rectangle();
        r1.setWidth(100);
        r1.setHeight(100);

        Rectangle r2 = new Rectangle();
        r2.xProperty().bind(r1.xProperty());
        r2.setY(110);
        r2.setWidth(100);
        r2.setHeight(100);

        final Rectangle r3 = new Rectangle();
        r1.xProperty().addListener(new DoublePropertyListener() {
            @Override public void changed() {
                r3.setX(r1.getX());
            }
        });
        r3.setY(220);
        r3.setWidth(100);
        r3.setHeight(100);

        stage.setScene(new Scene(800, 600, new Group(r1, r2, r3));
        stage.setVisible(true);

        // Animate the position of r1.x, which will cause r2.x and r3.x to update as well
        new Timeline(
            new KeyFrame(
                Duration.seconds(10),
                KeyValue.keyValue(r1.xProperty(), 700)
            )
        ).play();
    }
}
```

As you can see in the above code snippets, both defining and using the properties is very clean and elegant. The definition of the properties themselves is trivial (as much as JavaBeans has always been), while the usage of the pattern is very clean. Instead of adding listeners on the "bean", we add listeners directly to the property. Instead of relying on reflection for animations and binding, we use strongly typed Property instances. The getters and setters are still inline-able by HotSpot, and performance will rock since there is no boxing for primitive types. Boilerplate code is immensely reduced.

Various Performance Optimizations

However, the above implementation of the Rectangle class would by default require nearly 30 bytes for each double property (on a 32-bit VM). That is a fairly heavy penalty to pay. The good news is, we can trivially implement lazy properties, such that they would only be created when the DoubleProperty is read, or a value is set (but not when reading from the getter). To further demonstrate possibilities, suppose the "x" property defaulted to "100" instead of "0". This lazy implementation would still work without requiring any changes on the part of the client code:

Rectangle.java #2

```
public class Rectangle extends Shape {
    private DoubleProperty x;
    public double getX() { return x == null ? 100 : x.get(); }
    public void setX(double value) { xProperty().set(value); }
    public DoubleProperty xProperty() {
        return x == null ? (x = new DoubleProperty(100)) : x;
    }

    // other implementations here...
}
```

And in fact, the above pattern will result in **less** dynamic memory footprint than our current implementation, assuming the typical node is mostly left with its default values and with reasonably few properties bound. How so? In the current implementation, we reserve 2 fields per property -- one for the property value and one for a PropertyModel. For a double on a 32-bit JVM, that means 12 bytes when nobody is using the property, whereas with the above pattern, it means 4 bytes when nobody is using the property. Since most properties on most nodes are empty, then there are some reasonable savings to be made, while also cleaning up the source code and no longer requiring the use of Lombok.

This pattern works reliably well for the simple getter/setter case, but what about binding? Here the answer is somewhat more complicated and not entirely satisfying, yet notwithstanding this, it is a reasonable solution. Property classes both contain the actual field storage for the property as well as implement the binding. As such, it is relatively easy for them to implement lazy-binding semantics. In the getter for DoubleProperty, for example, it simply checks to see if it is bound. If so, return the value from the binding, otherwise, return its own locally stored value. When bound, the DoubleProperty can attach a listener such that when the binding changes, the DoubleProperty propagates that change onward to any listeners it also might have. It could even be more efficient, only installing a listener on the binding if somebody else is listening to it.

However, this is not enough. We have many places in our scene graph where we want to react to an invalidation caused either by somebody invoking the setter, or by a binding becoming invalid. There are some 397 such "on invalidate" triggers used in the 1.3 runtime source base. With the JavaFX property solution, there are really three possible options: subclassing, passing a method handle or using reflection. Fortunately, which to use is entirely up to the implementation, it is not exposed to end developers.

Here is a final look at Rectangle. This time I am also defining a special helper class which we could use to cut down on the number of anonymous classes for our code.

Rectangle.java #2

```
public class Rectangle extends Shape {
    private DoubleProperty x;
    public double getX() { return x == null ? 0 : x.get(); }
    public void setX(double value) { xProperty().set(value); }
    public DoubleProperty xProperty() {
        return x == null ? (x = new GeomDoubleProperty()) : x;
    }

    private DoubleProperty y;
    public double getY() { return y == null ? 0 : y.get(); }
    public void setY(double value) { yProperty().set(value); }
    public DoubleProperty yProperty() {
        return y == null ? (y = new GeomDoubleProperty()) : y;
    }

    // other implementations here...

    private final class GeomDoubleProperty extends DoubleProperty {
        @Override protected void invalidated() {
            impl_markDirty(dirtyBit);
            impl_geomChanged();
        }
    }
}
```

There are three downsides to this pattern. First, when "inflated", a property takes up more space than our previous **Enhanced** pattern, but only when the property is inflated due to a "set" call. When binding or listeners are involved, the costs for the **Enhanced** pattern and the **JavaFX** pattern are nearly identical.

Second, this pattern will require more classes, but only because we cannot take advantage of lambdas. Because we don't have lambda expressions (or even MethodHandles) in Java 6, we cannot implement our source code such that we use lightweight method references for the "on invalidate" functionality, but require a class of some kind, or reflection. One kills our static footprint, the other our runtime performance. Perhaps for a large number of rarely used fields using reflection is a possibility (which could dynamically use MethodHandles when running on Java 7), but for much of our runtime we will need subclasses (such as all the geometry & visual properties on shapes).

Third, this pattern does not provide for an easy way to listen to all property change (or invalidation) events which occur on an object. There are definitely solid use cases where this would be desired. Using an adapter or utility method that would use reflection to hook up listeners to all properties on a bean is possible, though it would mean inflating every property object, so you wouldn't want to do this on large swaths of objects in an application. However, since these use cases are fairly limited (and in every case I have considered it would be something a library would want to do rather than a typical developer), it seems a reasonable price to pay.

JavaFX Property Cook Book

Basic

Favors simplicity, runtime performance, and minimal static footprint for an increased dynamic footprint.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x = new DoubleProperty();
    public double getX() { return x.get(); }
    public void setX(double value) { x.set(value); }
    public DoubleProperty xProperty() { return x; }
}
```

Basic Half-Lazy

Favors runtime performance, minimal static footprint, and reasonable dynamic footprint for a somewhat more complicated implementation.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x;
    public double getX() { return x == null ? 0 : xProperty().get(); }
    public void setX(double value) { xProperty().set(value); }
    public DoubleProperty xProperty() {
        if (x == null) x = new DoubleProperty();
        return x;
    }
}
```

Basic Half-Lazy With Default Value

Favors runtime performance, minimal static footprint, and reasonable dynamic footprint for a somewhat more complicated implementation.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x;
    public double getX() { return x == null ? 100 : xProperty().get(); }
    public void setX(double value) { xProperty().set(value); }
    public DoubleProperty xProperty() {
        if (x == null) x = new DoubleProperty(100);
        return x;
    }
}
```

Basic Lazy

Favors runtime performance, minimal static footprint, and moderate dynamic footprint for a more complicated implementation.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x;
    private double _x;
    public double getX() { return x == null ? _x : x.get(); }
    public void setX(double value) {
        if (x == null) {
            _x = value;
        } else {
            x.set(value);
        }
    }
    public DoubleProperty xProperty() {
        if (x == null) x = new DoubleProperty(_x);
        return x;
    }
}
```

Basic Lazy With Default Value

Favors runtime performance, minimal static footprint, and moderate dynamic footprint for a more complicated implementation.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x;
    private double _x = 100;
    public double getX() { return x == null ? _x : x.get(); }
    public void setX(double value) {
        if (x == null) {
            _x = value;
        } else {
            x.set(value);
        }
    }
    public DoubleProperty xProperty() {
        if (x == null) x = new DoubleProperty(_x);
        return x;
    }
}
```

Basic Reacting To Changes

Can be used with any of the previous patterns (Basic, Basic Half-Lazy, Basic Lazy, and variants). This design favors larger static footprint and increased performance at runtime.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x = new DoubleProperty() {
        @protected void invalidated() {
            doSomethingAfterTheStoreButBeforeChangeListenersAreNotified();
        }
    };
    public double getX() { return x.get(); }
    public void setX(double value) { x.set(value); }
    public DoubleProperty xProperty() { return x; }
}
```

Reflection

Favors very minor static footprint and reduced class count for a somewhat more complicated implementation, somewhat slower runtime execution (on Java 6 this could be quite a large decrease in performance), and increased dynamic footprint.

Rectangle.java

```
public class Rectangle {
    private ReflectiveDoubleProperty x = new ReflectiveDoubleProperty("x");
    public double getX() { return x.read(); }
    public void setX(double value) { x.write(value); }
    public DoubleProperty xProperty() { return x; }
}
```

Reflection Half-Lazy

Favors very minor static footprint, reduced class count, and reasonable dynamic footprint for a somewhat more complicated implementation and decrease in runtime performance.

Rectangle.java

```
public class Rectangle {
    private ReflectiveDoubleProperty x;
    public double getX() { return x == null ? 0 : xProperty().read(); }
    public void setX(double value) { xProperty().write(value); }
    public DoubleProperty xProperty() {
        if (x == null) x = new ReflectiveDoubleProperty("x");
        return x;
    }
}
```

Reflection Half-Lazy With Default Value

Favors very minor static footprint, reduced class count, and reasonable dynamic footprint for a somewhat more complicated implementation and decrease in runtime performance.

Rectangle.java

```
public class Rectangle {
    private ReflectiveDoubleProperty x;
    public double getX() { return x == null ? 100 : xProperty().read(); }
    public void setX(double value) { xProperty().write(value); }
    public DoubleProperty xProperty() {
        if (x == null) x = new ReflectiveDoubleProperty("x", 100);
        return x;
    }
}
```

Reflection Lazy

Favors runtime performance, inconsequential static footprint, reduced class count, and reasonable dynamic footprint for a more complicated implementation.

Rectangle.java

```
public class Rectangle {
    private ReflectiveDoubleProperty x;
    private double _x;
    public double getX() { return x == null ? _x : x.read(); }
    public void setX(double value) {
        if (x == null) {
            _x = value;
        } else {
            x.write(value);
        }
    }
    public DoubleProperty xProperty() {
        if (x == null) x = new ReflectiveDoubleProperty("x", _x);
        return x;
    }
}
```

Reflection Lazy With Default Value

Favors runtime performance, inconsequential static footprint, reduced class count, and reasonable dynamic footprint for a more complicated implementation.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x;
    private double _x = 100;
    public double getX() { return x == null ? _x : x.read(); }
    public void setX(double value) {
        if (x == null) {
            _x = value;
        } else {
            x.write(value);
        }
    }
    public DoubleProperty xProperty() {
        if (x == null) x = new ReflectiveDoubleProperty("x", _x);
        return x;
    }
}
```

Reflection Reacting To Changes

Can be used with any of the previous reflection patterns (Reflection, Reflection Half-Lazy, Reflection Lazy, and variants). This design favors minimal static footprint for decreased runtime performance.

Rectangle.java

```
public class Rectangle {
    private ReflectiveDoubleProperty x = new ReflectiveDoubleProperty("x", "xInvalidated");
    private void xInvalidated() {
        doSomethingAfterXHasBeenChangedButBeforeListenersAreNotified();
    }
    public double getX() { return x.read(); }
    public void setX(double value) { x.write(value); }
    public DoubleProperty xProperty() { return x; }
}
```

Read-Only

A design which does not enforce callers to behave with dignity (they can up-cast to break encapsulation), but instead favors simplicity in design. This can be used for any of the previous patterns (Basic or Reflection). Simply modify the setter to the access you require, and modify the getter to return a ReadOnly property of your choice.

Rectangle.java

```
/**
 * Allows the "x" property to be changed by this class, but not by external classes or sub classes.
 */
public class Rectangle {
    private DoubleProperty x = new DoubleProperty();
    public ReadOnlyDoubleProperty xProperty() { return x; }
    public double getX() { return x.get(); }
    private void setX(double value) { x.set(value); }
}
```

Computed

A computed property in our system is simply one that is constructed via a binding. This is necessary because we must have a Property object to which clients can register listeners, and thus using binding is the least error-prone way to ensure change events are fired appropriately.

Rectangle.java

```
public class Rectangle {
    private DoubleProperty x = new DoubleProperty();
    private DoubleProperty width = new DoubleProperty();

    private DoubleProperty x2 = new DoubleProperty(); {
        x2.bind(x.add(width));
    }
    public ReadOnlyDoubleProperty x2Property() { return x2; }
}
```