Chapter 15

Using Properties to Create Dynamic Scenes

In This Chapter

- Introducing and creating properties
- Working with read-only and read/write properties
- Adding event listeners to your properties
- Binding properties together

In this chapter, you discover how to use a powerful feature of JavaFX — properties. Simply put, a JavaFX *property* is an observable value that's exposed by a class. Properties are observable in the sense that you can attach listeners to them. These listeners can be invoked whenever the value of the property changes or becomes unknown.

One of the best features of properties is that you can *bind* to them, or connect properties together so that when one property changes, the other property is adjusted automatically. In other words, binding allows two properties to be synchronized. When one property changes, the other property changes as well.

In this chapter, I discuss the basics of working with properties. First, you read about how to create your own properties. Then, you figure out how to attach listeners to properties so your code can respond when the property value changes or becomes invalid. And finally, I tell you how to bind.

Introducing JavaFX Properties

In object-oriented programming parlance, a *property* is a value that represents the state of an instantiated object which can be retrieved and in some cases set by users of the object. In some object-oriented programming languages, such as C#, the concept of properties is built in to the language. Alas, such is not the case with Java. Java has no built-in features for implementing properties . . . at least, not until now.

Prior to Java 8, Java developers usually followed the pattern of using property getters and setters to retrieve and set property values. A property *getter* is a public method that retrieves the value of a property, and a *setter* is a public method that sets the value of a property.

For example, suppose you're creating a class that represents a customer, and each customer is identified by a unique customer number. You might store the customer number internally as a private field named `customerNumber`. Then, you'd provide a public method named `getCustomerNumber` to return the customer's number and another public method named `setCustomerNumber` to set the customer number. The resulting code would look like this:

```
class Customer
{
    private int customerNumber;

    public int getCustomerNumber()
    {
        return customerNumber;
    }

    public void setCustomerNumber(int value)
    {
        customerNumber = value;
    }
}
```

This pattern of using getter and setter methods has a name: the *accessor pattern.* (No, this has nothing to do with property taxes. Property taxes are determined by an *assessor,* not an *accessor.*)

JavaFX 8 introduces a new scheme for implementing properties, which dutifully follows the accessor pattern. As a result, JavaFX classes that implement properties must provide a getter and a setter that returns and sets the value of the properties (unless the property is read-only, in which case only a `get` method is required).

However, instead of using a simple field to represent the value of the property, JavaFX uses special property classes to represent properties. These classes encapsulate the value of a property in an object that provides the new whiz-bang features that enable you to listen for changes in the property value or bind properties together.

When JavaFX properties are used, the getter and setter methods return the value that's encapsulated by the property object. In addition to the getter and setter methods, JavaFX properties introduce a third method which returns the actual property itself. This allows users of the property to directly access the property object, which in turn lets them access those new whiz-bang features.

Naming conventions are an essential aspect of using JavaFX properties correctly. Every property has a name, which by convention begins with a lowercase letter. For example, a property that represents a person's first name might be called `firstName`. The getter and setter methods are created by capitalizing the property name and prefixing it with the word `get` or `set`. The method that returns the property object is the name of the property (uncapitalized) followed by the word `Property`.

Thus, a class that implements a read/write property named `firstName` must expose three methods:

```
getFirstName

setFirstName

firstNameProperty
```

If the property is read-only, the `setFirstName` method would be omitted.

Java API Properties

All the JavaFX API classes that are presented in this book make extensive use of properties. In fact, just about every API `get` or `set` method is actually a property getter or setter, and has a corresponding method that returns the property itself.

For example, consider the `TextField` class, which has methods named `getText` and `setText` that get and set the text contained in the text field. These methods are actually getters and setters for a property named `text`, and the `TextField` class has a method named `textProperty` that provides direct access to the `text` property.

Likewise with the `HBox` class: It has properties named `alignment`, `hgrow`, `padding`, and `spacing` which correspond to the `getAlignment`, `setAlignment`, `getHgrow`, `setHgrow`, `getPadding`, `setPadding`, `getSpacing`, and `setSpacing` methods.

So far in this book, you've had no need to access the properties directly, so you've relied on the getter and setter methods to manipulate the property values. Later in this chapter, when you discover how to bind property values, you see that accessing these properties can be very useful.

In the next few sections, I discuss how to create your own JavaFX properties. Even if you never create JavaFX properties for your own classes, knowing how to do so will help you understand the benefits of working with the properties that are defined as part of the standard JavaFX API classes.

JavaFX Property Classes

At the heart of JavaFX properties is a collection of classes that create property objects. There are a lot of them, as JavaFX provides four important classes for each of its basic data types, and property classes are provided for ten different data types. Do the math: That means there are 40 property classes. The following paragraphs describe the four classes for `String` properties:

- **`ReadOnlyStringProperty`:** An abstract class that represents a read-only property whose value can be read but not modified.
- **`StringProperty`:** Another abstract class that represents a read-write property. This class extends `ReadOnlyStringProperty`.
- **`SimpleStringProperty`:** This is the class that you instantiate to create a read/write string property.
- **`ReadOnlyStringWrapper`:** This is the class you instantiate to create a read-only string property. The use of this class is a bit confusing, so I won't explain it quite yet. You see how it works in the section "Creating a Read-Only Property" later in this chapter.

For your reference, Table 15-1 lists all 40 of the classes used to create properties of the various types.

Table 15-1 JavaFX Property Classes

Boolean Classes	***Long Classes***
`ReadOnlyBooleanProperty`	`ReadOnlyLongProperty`
`BooleanProperty`	`LongProperty`
`SimpleBooleanProperty`	`SimpleLongProperty`
`ReadOnlyBooleanWrapper`	`ReadOnlyLongWrapper`
Double Classes	***Map***
`ReadOnlyDoubleProperty`	`ReadOnlyMapProperty<K,V>`
`DoubleProperty`	`MapProperty<K,V>`
`SimpleDoubleProperty`	`SimpleMapProperty<K,V>`
`ReadOnlyDoubleWrapper`	`ReadOnlyMapWrapper<K,V>`
Float Classes	***Object Classes***
`ReadOnlyFloatProperty`	`ReadOnlyObjectProperty<T>`
`FloatProperty`	`ObjectProperty<T>`
`SimpleFloatProperty`	`SimpleObjectProperty<T>`
`ReadOnlyFloatWrapper`	`ReadOnlyObjectWrapper<T>`
Integer Classes	***Set Classes***
`ReadOnlyIntegerProperty`	`ReadOnlySetProperty<E>`
`IntegerProperty`	`SetProperty<E>`
`SimpleIntegerProperty`	`SimpleSetProperty<E>`
`ReadOnlyIntegerWrapper`	`ReadOnlySetWrapper<E>`
List Classes	***String Classes***
`ReadOnlyListProperty<E>`	`ReadOnlyStringProperty`
`ListProperty<E>`	`StringProperty`
`SimpleListProperty<E>`	`SimpleStringProperty`
`ReadOnlyListWrapper<E>`	`ReadOnlyStringWrapper`

Note that four of the types shown in Table 15-1 — `List`, `Map`, `Object`, and `Set` — are generic. For the `List` and `Set` classes, you must specify the element type for the underlying list and set collections; for the `Map` type, you need to specify types for the keys and values. The `Object` property classes let you create properties of any type you wish, but you must specify the type so that JavaFX can enforce type safety.

Creating a Read/Write Property

To create a basic property whose value can be read and written, you need to use two of the classes for the property type: the property class of the correct type and the corresponding simple property. For example, to create a property of type `Double`, you need to use both the `DoubleProperty` class and the `SimpleDoubleProperty` class.

Here are the steps to create a read/write property:

1. **Create a local field for the property using the property class for the correct type.**

   The field should be defined with `private` visibility, and it should be `final`. For example:

   ```
   private String Property firstName;
   ```

2. **Create an instance of the property using the simple property class of the correct type.**

   The constructor for the simple property type accepts three parameters, representing the object that contains the property (usually specified as `this`, a string that represents the name of the property, and the property's default value). For example:

   ```
   firstName = new SimpleStringProperty(this,
       "firstName", "");
   ```

   Here, `this` is specified as the containing object, `firstName` is the name of the property, and the default value is an empty string.

   TIP

   It is often convenient to declare the private property field and instantiate the property in the same statement, like this:

   ```
   StringProperty firstName =
       new SimpleStringProperty(this,
           "firstName", "");
   ```

3. **Create a getter for the property.**

   The getter method name should be `public` or `protected`, it should be `final`, it should follow the property naming convention (`get` followed by the name of the property with an initial cap), and it should return a value of the underlying property type. It should then call the private property's `get` method to retrieve the value of the property, like this:

   ```
   public final String getFirstName
   {
       return firstName.get();
   }
   ```

4. **Create a setter for the property.**

   The setter method name should by `public` or `protected`, it should be `final`, it should follow the property naming convention (`set` followed by the name of the property with an initial cap), and it should accept a parameter value of the underlying property type. It should then call the private property's `set` method to set the property to the passed value. For example:

   ```
   public final void setFirstName(String value)
   {
       firstName.set(value);
   }
   ```

5. **Create the property accessor.**

   This method should return the property object itself:

   ```
   public final StringProperty firstNameProperty()
   {
       return firstName;
   }
   ```

   Notice that the type is `StringProperty`, not `SimpleStringProperty`.

6. **Repeat the entire procedure for every property in your class.**

Here's a complete example that implements a read/write property named `firstName` in a class named `Customer`:

```
public class Customer
{
    StringProperty firstName =
        new SimpleStringProperty(this,
            "firstName", "");

    public final String getFirstName
    {
        return firstName.get();
    }

    public final void setFirstName(String value)
    {
        firstName.set(value);
    }

    public final StringProperty firstNameProperty()
    {
        return firstName;
    }
}
```

Creating a Read-Only Property

Although a read-only property has less functionality than a read/write property, it's actually more complicated to implement. Why? Because internally — within the class that contains the read-only property — you need to be able to read or write the value of the property. But externally — that is, outside of the class that defines the read-only property — you must ensure that users can read but not write the property value.

You might think that omitting the setter method would be enough to create a read-only property. But the problem is that in addition to getter and setter methods, JavaFX properties also expose a property accessor method that provides direct access to the property object itself.

WARNING!

The following is an example of how *not* to create a read-only property:

```
StringProperty firstName =
    new SimpleStringProperty(this,
        "firstName", "");

public final String getFirstName()
{
    return firstName.get();
}

public final StringProperty firstNameProperty()
{
    return firstName;
}
```

This code is the same as the code used to create a read/write property in the preceding section, except that I omitted the `setFirstName` method. Unfortunately, this property definition does not prevent users of the class that defines the property from modifying the property. To do so, all the user would need to do is access the property and then call the property's `set` method directly.

For example, suppose this property is part of a class named `Customer`, an instance of which is referenced by the variable `cust`. The following code would set the value of the read-only property:

```
cust.getFirstName().set("Bogus Value");
```

To safely create a read-only property, you must actually create two copies of the property: a read-only version and a read/write version. The read-only version will be exposed to the outside world. The read/write version will be used internally, within the class that defines the property. Then, you must synchronize these two properties so that whenever the value of the internal read/write property changes, the value of the external read-only property is updated automatically.

To accomplish this, JavaFX provides two additional classes for each property data type: a read-only property class and a read-only wrapper class. The read-only property class is the one you share with the outside world via the property accessor method. The read-only wrapper class is the one you use to create the private field used to reference the property within the program.

Here's a complete example that implements a read-only integer property named `customerNumber` in a class named `Customer`:

```
public class Customer
{
    ReadOnlyIntegerWrapper customerNumber =
        new ReadOnlyIntegerWrapper(this,
            "customerNumber", 0);

    public final Integer getCustomerNumber()
    {
        return customerNumber.get();
    }

    public final ReadOnlyIntegerProperty()
        customerNumberProperty()
    {
        return customerNumber.getReadOnlyProperty();
    }

    // more class details go here

}
```

The key to understanding how this works is realizing that the read-only wrapper class is an extension of the simple property class which adds just one new method: `getReadOnlyProperty`, which returns a read-only copy of the simple property. This read-only copy is automatically synchronized with the simple property, so that whenever a change is made to the underlying simple property, the value of the read-only property will be changed as well.

Creating Properties More Efficiently

The advanced capabilities of JavaFX properties, which you'll come to appreciate in the final sections of this chapter, do not come without a cost. Specifically, instantiating a property object takes more memory and processing time than creating a simple field-based property. And in many classes, the advanced capabilities of a JavaFX property object are only occasionally needed. This, instantiating property objects for every property in a class whether the object is needed or not, is wasteful.

In this section, I show you a technique for creating properties in which the property objects themselves are not instantiated until the property accessor itself is called. That way, the property object is not created unless it is actually needed. Here are the details of this technique:

1. **Declare a private field to hold the data represented by the property.**

   For example, for a string property, you create a `String` variable. For the variable name, use the name of the property prefixed by an underscore, like this:

   ```
   private final String _firstName = "";
   ```

2. **Create, but do not instantiate, a private variable to represent the property object.**

   In other words, declare the variable but do not call the class constructor:

   ```
   private final SimpleStringProperty firstName;
   ```

3. **Create the getter.**

   In the getter, use an `if` statement to determine whether the property object exists. If it does, return the value from the property. If it doesn't, return the value of the private field. For example:

   ```
   public final String getFirstName()
   {
       if (firstName == null)
           return _firstName;
       else
           return firstName.get();
   }
   ```

4. **Create the setter.**

   Use the same technique in the setter:

```
public final void setFirstName(String value)
{
    if (firstName == null)
        _firstName = value;
    else
        firstName.set(value);
}
```

5. **Create the property accessor.**

   In this method, first check whether the property object exists and create the object if it does not exist. Use the value of the private field as the initial value of the property. Then, return the object:

```
public final StringProperty firstNameProperty()
{if (firstName == null)
        firstName = new SimpleStringProperty(
            this, "firstName", _firstName);
   return firstName;
}
```

Here's what it looks like put together in a class named `Customer`:

```
Public class Customer
{
    private final String _firstName = "";
    private final SimpleStringProperty firstName;

    public final String getFirstName()
    {
        if (firstName == null)
            return _firstName;
        else
            return firstName.get();
    }

    public final void setFirstName(String value)
    {
        if (firstName == null)
            _firstName = value;
        else
            firstName.set(value);
    }

    public final StringProperty firstNameProperty()

    {          if (firstName == null)
            firstName = new SimpleStringProperty(
                this, "firstName", _firstName);
       return firstName;
    }
}
```

Using Property Events

JavaFX properties provide an `addListener` method that lets you add event handlers that are called whenever the value of a property changes. You can create two types of property event handlers:

- A **change listener,** which is called whenever the value of the property has been recalculated. The change listener is passed three arguments: the property whose value has changed, the previous value of the property, and the new value.
- An **invalidation listener,** which is called whenever the value of the property becomes unknown. This event is raised when the value of the property needs to be recalculated, but has not yet been recalculated. An invalidation event listener is passed just one argument: the property object itself.

Change and invalidation listeners are defined by functional interfaces named `ChangeListener` and `InvalidationListener`. Because these interfaces are functional interfaces, you can use Lambda expressions to implement them. Here's how you use a Lambda expression to register a change listener on the `text` property of a text field named `text1`:

```
text1.textProperty().addListener(
    (observable, oldvalue, newvalue) ->
        // code goes here
    );
```

Here's an example that registers an invalidation listener:

```
text1.textProperty().addListener(
    (observable) ->
        // code goes here
    );
```

The only way the `addListener` knows whether you are registering a change listener or an invalidation listener is by looking at the arguments you specify for the Lambda expression. If you provide three arguments, `addListener` registers a change listener. If you provide just one argument, an invalidation listener is installed.

Listing 15-1 shows a simple JavaFX application that uses change listeners to vary the size of a rectangle automatically with the size of the stack pane that contains it. A change listener is registered with the stack pane's width property so that whenever the width of the stack pane changes, the width of the rectangle is automatically set to half the new width of the stack pane. Likewise, a change listener is registered on the height property to change the rectangle's height.

Figure 15-1 shows this application in action. This figure shows the initial window displayed by the application as well as how the window appears after the user has made the window taller and wider. As you can see, the rectangle has increased in size proportionately.

Listing 15-1: The Auto Rectangle Program

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.shape.*;
import javafx.scene.paint.*;

public class AutoRectangle extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    @Override public void start(Stage primaryStage)
    {

        Rectangle r = new Rectangle(100,100);                  →18
        r.setFill(Color.LIGHTGRAY);
        r.setStroke(Color.BLACK);
        r.setStrokeWidth(2);

        StackPane p = new StackPane();                         →23
        p.setMinWidth(200);
        p.setPrefWidth(200);
        p.setMaxWidth(200);
        p.setMinHeight(200);
```

(continued)

Listing 15-1 ***(continued)***

```
        p.setPrefHeight(200);
        p.setMaxHeight(200);

        p.getChildren().add(r);

        p.widthProperty().addListener(                                          →33
            (observable, oldvalue, newvalue) ->
                  r.setWidth((Double)newvalue/2)
            );

        p.heightProperty().addListener(                                         →38
            (observable, oldvalue, newvalue) ->
                  r.setHeight((Double)newvalue/2)
            );

        Scene scene = new Scene(p);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Auto Rectangle");
        primaryStage.show();
    }

}
```

The following paragraphs describe the highlights:

- → **18:** These lines create a 100x100 rectangle and set the rectangle's fill color, stroke color, and stroke width.
- → **23:** These lines create a stack pane and set its width and height properties.
- → **33:** These lines use a Lambda expression to register a change handler with the stack pane's width parameter. When the stack pane's width changes, the width of the rectangle is set to one half of the stack pane's width.
- → **38:** These lines use a Lambda expression to register a change handler with the stack pane's height parameter. When the stack pane's height changes, the height of the rectangle is set to one half of the stack pane's height.

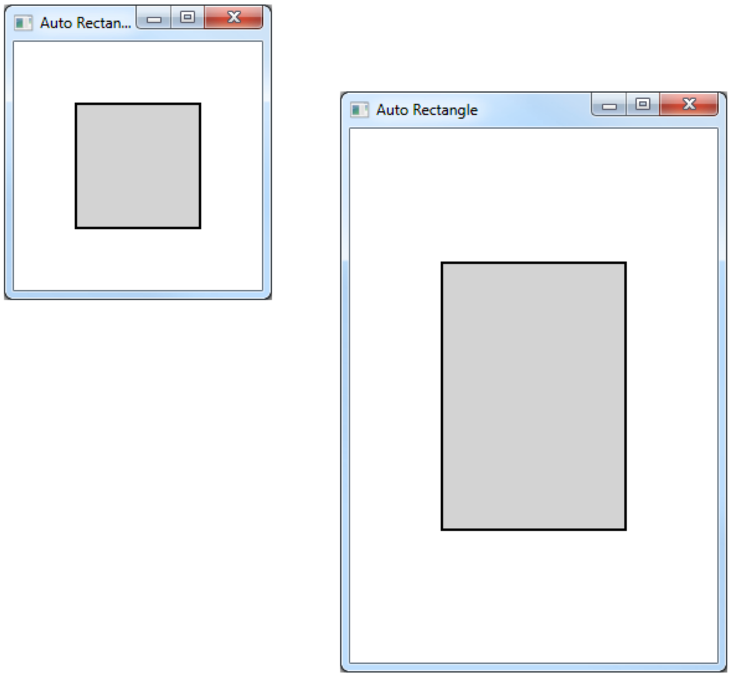


Figure 15-1: The Auto Rectangle program in action.

Binding Properties

JavaFX *property binding* allows you to synchronize the value of two properties so that whenever one of the properties changes, the value of the other property is updated automatically. Two types of binding are supported:

- **Unidirectional binding:** With unidirectional binding, the binding works in just one direction. For example, if you bind property A to property B, the value of property A changes when property B changes, but not the other way around.
- **Bidirectional binding:** With bidirectional binding, the two property values are synchronized so that if either property changes, the other property is automatically changed as well.

Setting up either type of binding is surprisingly easy. Every property has a `bind` and a `bindBiDirectional` method. To set up a binding, simply call this method, specifying the property you want to bind to as the argument.

Here's an example that creates a unidirectional binding on the `text` property of a label to the `text` property of a text field, so that the contents of the label always displays the contents of the text field:

```
lable1.textProperty().bind(text1.textProperty());
```

With this binding in place, the text displayed by `label1` is automatically updated, character by character, when the user types data into the text field.

The following example shows how to create a bidirectional binding between two text fields, named `text1` and `text2`:

```
text1.textProperty()
      .bindBidirectional(text2.textProperty());
```

With this binding in place, any text you type into either text field will be replicated automatically in the other.

To show how binding can be used in a complete program, Listing 15-2 shows a program with two text fields with a pair of labels bound to each. The first text field accepts the name of a character in a play, and the second text field accepts the name of an actor. The labels display the actor who will play the role, as shown in Figure 15-2.

Listing 15-2: The Role Player Program

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.geometry.*;
import javafx.scene.control.*;

public class RolePlayer extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    TextField txtCharacter;
    TextField txtActor;

    @Override public void start(Stage primaryStage)
    {

        // Create the Character label
        Label lblCharacter = new Label("Character's Name:");
        lblCharacter.setMinWidth(100);
        lblCharacter.setAlignment(Pos.BOTTOM_RIGHT);

        // Create the Character text field
        txtCharacter = new TextField();
        txtCharacter.setMinWidth(200);
        txtCharacter.setMaxWidth(200);
        txtCharacter.setPromptText("Enter the name of the character here.");

        // Create the Actor label
        Label lblActor = new Label("Actor's Name:");
        lblActor.setMinWidth(100);
        lblActor.setAlignment(Pos.BOTTOM_RIGHT);

        // Create the Actor text field
        txtActor = new TextField();
        txtActor.setMinWidth(200);
        txtActor.setMaxWidth(200);
        txtActor.setPromptText("Enter the name of the actor here.");

        // Create the Role labels
        Label lblRole1 = new Label("The role of ");
        Label lblRole2 = new Label();
        Label lblRole3 = new Label(" will be played by ");
        Label lblRole4 = new Label();

        // Create the Character pane
        HBox paneCharacter = new HBox(20, lblCharacter, txtCharacter);
        paneCharacter.setPadding(new Insets(10));

        // Create the Actor pane
        HBox paneActor = new HBox(20, lblActor, txtActor);
        paneActor.setPadding(new Insets(10));

        // Create the Role pane
        HBox paneRole = new HBox(lblRole1, lblRole2, lblRole3, lblRole4);
        paneRole.setPadding(new Insets(10));
```

(continued)

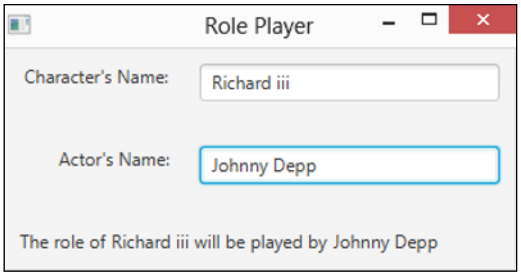
Listing 15-2 *(continued)*

```
        // Add the Character and Actor panes to a VBox
        VBox pane = new VBox(10, paneCharacter, paneActor, paneRole);

        // Create the bindings
        lblRole2.textProperty().bind(txtCharacter.textProperty());
        lblRole4.textProperty().bind(txtActor.textProperty());


        // Set the stage
        Scene scene = new Scene(pane);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Role Player");
        primaryStage.show();    }

}
```



Character's Name: Richard iii

Actor's Name: Johnny Depp

The role of Richard iii will be played by Johnny Depp

Figure 15-2: The Role Player application in action.