CHAPTER 2

Properties and Bindings

In this chapter, you will learn:

- What a property is in JavaFX
- How to create a property object and use it
- The class hierarchy of properties in JavaFX
- How to handle the invalidation and change events in a property object
- What a binding is in JavaFX and how to use unidirectional and bidirectional bindings
- About the high-level and low-level binding APIs in JavaFX

This chapter discusses the properties and binding support in Java and JavaFX. If you have experience using the JavaBeans API for properties and binding, you can skip the first few sections, which discuss the properties and binding support in Java, and start with the section "Understanding Properties in JavaFX."

The examples of this chapter lie in the `com.jdojo.binding` package. In order for them to work, you must add a corresponding line to the `module-info.java` file:

```
...
opens com.jdojo.binding to javafx.graphics, javafx.base;
...
```

What Is a Property?

A Java class can contain two types of members: *fields* and *methods*. Fields represent the state of objects, and they are declared private. Public methods, known as *accessors*, or *getters* and *setters*, are used to read and modify private fields. In simple terms, a Java class that has public accessors, for all or part of its private fields, is known as a Java *bean*, and the accessors define the properties of the bean. Properties of a Java bean allow users to customize its state, behavior, or both.

Java beans are observable. They support property change notification. When a public property of a Java bean changes, a notification is sent to all interested listeners.

In essence, Java beans define reusable components that can be assembled by a builder tool to create a Java application. This opens the door for third parties to develop Java beans and make them available to others for reuse.

A property can be read-only, write-only, or read/write. A ready-only property has a getter but no setter. A write-only property has a setter but no getter. A read/write property has a getter and a setter.

K. Sharan and P. Späth, *Learn JavaFX 17*, https://doi.org/10.1007/978-1-4842-7848-2_2

Java IDEs and other builder tools (e.g., a GUI layout builder) use introspection to get the list of properties of a bean and let you manipulate those properties at design time. A Java bean can be visual or nonvisual. Properties of a bean can be used in a builder tool or programmatically.

The JavaBeans API provides a class library, through the `java.beans` package, and naming conventions to create and use Java beans. The following is an example of a `Person` bean with a read/write `name` property. The `getName()` method (the getter) returns the value of the `name` field. The `setName()` method (the setter) sets the value of the `name` field:

```
 // Person.java
package com.jdojo.binding;

public class Person {
        private String name;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
}
```

By convention, the names of the getter and setter methods are constructed by appending the name of the property, with the first letter in uppercase, to the words *get* and *set*, respectively. The getter method should not take any parameters, and its return type should be the same as the type of the field. The setter method should take a parameter whose type should be the same as the type of the field, and its return type should be `void`.

The following snippet of code manipulates the `name` property of a `Person` bean programmatically:

```
Person p = new Person();
p.setName("John Jacobs");
String name = p.getName();
```

Some object-oriented programming languages, for example, C#, provide a third type of class member known as a *property*. A property is used to read, write, and compute the value of a private field from outside the class. C# lets you declare a `Person` class with a `Name` property as follows:

```
// C# version of the Person class
public class Person {
        private string name;

        public string Name {
                get { return name; }
                set { name = value; }
        }
}
```

In C#, the following snippet of code manipulates the `name` private field using the `Name` property; it is equivalent to the previously shown Java version of the code:

```
Person p = new Person();
p.Name = "John Jacobs";
string name = p.Name;
```

If the accessors of a property perform the routine work of returning and setting the value of a field, C# offers a compact format to define such a property. You do not even need to declare a private field in this case. You can rewrite the `Person` class in C# as shown here:

```
// C# version of the Person class using the compact format
public class Person {
        public string Name { get; set; }
}
```

So, what is a property? A *property* is a publicly accessible attribute of a class that affects its state, behavior, or both. Even though a property is publicly accessible, its use (read/write) invokes methods that hide the actual implementation to access the data. Properties are observable, so interested parties are notified when its value changes.

■ **Tip** In essence, properties define the public state of an object that can be read, written, and observed for changes. Unlike other programming languages, such as C#, properties in Java are not supported at the language level. Java support for properties comes through the JavaBeans API and design patterns. For more details on properties in Java, please refer to the JavaBeans specification, which can be downloaded from `www.oracle.com/java/technologies/javase/javabeans-spec.html`.

Apart from simple properties, such as the `name` property of the `Person` bean, Java also supports *indexed*, *bound*, and *constrained* properties. An indexed property is an array of values that are accessed using indexes. An indexed property is implemented using an array data type. A bound property sends a notification to all listeners when it is changed. A constrained property is a bound property in which a listener can veto a change.

What Is a Binding?

In programming, the term *binding* is used in many different contexts. Here, I want to define it in the context of *data binding*. Data binding defines a relation between data elements (usually variables) in a program to keep them synchronized. In a GUI application, data binding is frequently used to synchronize the elements in the data model with the corresponding UI elements.

Consider the following statement, assuming that x, y, and z are numeric variables:

```
x = y + z;
```

The preceding statement defines a binding between x, y, and z. When it is executed, the value of x is synchronized with the sum of y and z. A binding also has a time factor. In the preceding statement, the value of x is bound to the sum of y and z and is valid at the time the statement is executed. The value of x may not be the sum of y and z before and after the preceding statement is executed.

Sometimes, it is desired for a binding to hold over a period. Consider the following statement that defines a binding using `listPrice`, `discounts`, and `taxes`:

```
soldPrice = listPrice - discounts + taxes;
```

For this case, you would like to keep the binding valid forever, so the sold price is computed correctly, whenever `listPrice`, `discounts`, or `taxes` change.

In the preceding binding, `listPrice`, `discounts`, and `taxes` are known as *dependencies*, and it is said that `soldPrice` is bound to `listPrice`, `discounts`, and `taxes`.

For a binding to work correctly, it is necessary that the binding is notified whenever its dependencies change. Programming languages that support binding provide a mechanism to register listeners with the dependencies. When dependencies become invalid or they change, all listeners are notified. A binding may synchronize itself with its dependencies when it receives such notifications.

A binding can be an *eager binding* or a *lazy binding*. In an eager binding, the bound variable is recomputed immediately after its dependencies change. In a lazy binding, the bound variable is not recomputed when its dependencies change. Rather, it is recomputed when it is read the next time. A lazy binding performs better compared to an eager binding.

A binding may be *unidirectional* or *bidirectional*. A unidirectional binding works only in one direction; changes in the dependencies are propagated to the bound variable. A bidirectional binding works in both directions. In a bidirectional binding, the bound variable and the dependency keep their values synchronized with each other. Typically, a bidirectional binding is defined only between two variables. For example, a bidirectional binding, x = y and y = x, declares that the values of x and y are always the same.

Mathematically, it is not possible to define a bidirectional binding between multiple variables uniquely. In the preceding example, the sold price binding is a unidirectional binding. If you want to make it a bidirectional binding, it is not uniquely possible to compute the values of the list price, discounts, and taxes when the sold price is changed. There are an infinite number of possibilities in the other direction.

Applications with GUIs provide users with UI widgets, for example, text fields, check boxes, and buttons, to manipulate data. The data displayed in UI widgets have to be synchronized with the underlying data model and vice versa. In this case, a bidirectional binding is needed to keep the UI and the data model synchronized.

Understanding Binding Support in JavaBeans

Before I discuss JavaFX properties and binding, let's take a short tour of binding support in the JavaBeans API. You may skip this section if you have used the JavaBeans API before.

Java has supported binding of bean properties since its early releases. Listing 2-1 shows an `Employee` bean with two properties, `name` and `salary`.

Listing 2-1. An Employee Java Bean with Two Properties Named `name` and `salary`

```
// Employee.java
package com.jdojo.binding;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Employee {
        private String name;
        private double salary;
        private PropertyChangeSupport pcs = new PropertyChangeSupport(this);
```

```
        public Employee() {
                this.name = "John Doe";
                this.salary = 1000.0;
        }

        public Employee(String name, double salary) {
                this.name = name;
                this.salary = salary;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public double getSalary() {
                return salary;
        }

        public void setSalary(double newSalary) {
                double oldSalary = this.salary;
                this.salary = newSalary;

                // Notify the registered listeners about the change
                pcs.firePropertyChange("salary", oldSalary, newSalary);
        }

        public void addPropertyChangeListener(
                        PropertyChangeListener listener) {
                pcs.addPropertyChangeListener(listener);
        }

        public void removePropertyChangeListener(
                        PropertyChangeListener listener) {
                pcs.removePropertyChangeListener(listener);
        }

        @Override
        public String toString() {
                return "name = " + name + ", salary = " + salary;
        }
}
```

Both properties of the `Employee` bean are read/write. The `salary` property is also a bound property. Its setter generates property change notifications when the salary changes.

Interested listeners can register or deregister for the change notifications using the `addPropertyChangeListener()` and `removePropertyChangeListener()` methods. The `PropertyChangeSupport` class is part of the JavaBeans API that facilitates the registration and removal of property change listeners and firing of the property change notifications.

Any party interested in synchronizing values based on the salary change will need to register with the `Employee` bean and take necessary actions when it is notified of the change.

Listing 2-2 shows how to register for salary change notifications for an `Employee` bean. The output below it shows that salary change notification is fired only twice, whereas the `setSalary()` method is called three times. This is true because the second call to the `setSalary()` method uses the same salary amount as the first call, and the `PropertyChangeSupport` class is smart enough to detect that. The example also shows how you would bind variables using the JavaBeans API. The tax for an employee is computed based on a tax percentage. In the JavaBeans API, property change notifications are used to bind the variables.

Listing 2-2. An EmployeeTest Class That Tests the Employee Bean for Salary Changes

```
// EmployeeTest.java
package com.jdojo.binding;

import java.beans.PropertyChangeEvent;

public class EmployeeTest {
        public static void main(String[] args) {
                final Employee e1 = new Employee("John Jacobs", 2000.0);

                // Compute the tax
                computeTax(e1.getSalary());

                // Add a property change listener to e1
                e1.addPropertyChangeListener(
                         EmployeeTest::handlePropertyChange);

                // Change the salary
                e1.setSalary(3000.00);
                e1.setSalary(3000.00); // No change notification is sent.
                e1.setSalary(6000.00);
        }

        public static void handlePropertyChange(PropertyChangeEvent e) {
                String propertyName = e.getPropertyName();

                if ("salary".equals(propertyName)) {
                        System.out.print("Salary has changed. ");
                        System.out.print("Old:" + e.getOldValue());
                        System.out.println(", New:" +
                                    e.getNewValue());
                        computeTax((Double)e.getNewValue());
                }
        }

        public static void computeTax(double salary) {
                final double TAX_PERCENT = 20.0;
                double tax = salary * TAX_PERCENT/100.0;
                System.out.println("Salary:" + salary + ", Tax:" + tax);
        }
}
```

```
Salary:2000.0, Tax:400.0
Salary has changed. Old:2000.0, New:3000.0
Salary:3000.0, Tax:600.0
Salary has changed. Old:3000.0, New:6000.0
Salary:6000.0, Tax:1200.0
```

Understanding Properties in JavaFX

JavaFX supports properties, events, and binding through *properties* and *binding* APIs. Properties support in JavaFX is a huge leap forward from the JavaBeans properties.

All properties in JavaFX are observable. They can be observed for invalidation and value changes. There can be read/write or read-only properties. All read/write properties support binding.

In JavaFX, a property can represent a value or a collection of values. This chapter covers properties that represent a single value. I will cover properties representing a collection of values in Chapter 3.

In JavaFX, properties are objects. There is a property class hierarchy for each type of property. For example, the `IntegerProperty`, `DoubleProperty`, and `StringProperty` classes represent properties of `int`, `double`, and `String` types, respectively. These classes are `abstract`. There are two types of implementation classes for them: one to represent a read/write property and one to represent a wrapper for a read-only property. For example, the `SimpleDoubleProperty` and `ReadOnlyDoubleWrapper` classes are concrete classes whose objects are used as read/write and read-only double properties, respectively.

The following is an example of how to create an `IntegerProperty` with an initial value of 100:

```
IntegerProperty counter = new SimpleIntegerProperty(100);
```

Property classes provide two pairs of getter and setter methods: `get()`/`set()` and `getValue()`/`setValue()`. The `get()` and `set()` methods get and set the value of the property, respectively. For primitive type properties, they work with primitive type values. For example, for `IntegerProperty`, the return type of the `get()` method and the parameter type of the `set()` method are `int`. The `getValue()` and `setValue()` methods work with an object type; for example, their return type and parameter type are `Integer` for `IntegerProperty`.

■ **Tip** For reference type properties, such as `StringProperty` and `ObjectProperty<T>`, both pairs of getter and setter work with an object type. That is, both `get()` and `getValue()` methods of `StringProperty` return a `String`, and `set()` and `setValue()` methods take a `String` parameter. With autoboxing for primitive types, it does not matter which version of getter and setter is used. The `getValue()` and `setValue()` methods exist to help you write generic code in terms of object types.

The following snippet of code uses an `IntegerProperty` and its `get()` and `set()` methods. The `counter` property is a read/write property as it is an object of the `SimpleIntegerProperty` class:

```
IntegerProperty counter = new SimpleIntegerProperty(1);
int counterValue = counter.get();
System.out.println("Counter:" + counterValue);
```

```
counter.set(2);
counterValue = counter.get();
System.out.println("Counter:" + counterValue);
```

```
Counter:1
Counter:2
```

Working with read-only properties is a bit tricky. A `ReadOnlyXXXWrapper` class wraps two properties of `XXX` type: one read-only and one read/write. Both properties are synchronized. Its `getReadOnlyProperty()` method returns a `ReadOnlyXXXProperty` object.

The following snippet of code shows how to create a read-only `Integer` property. The `idWrapper` property is read/write, whereas the `id` property is read-only. When the value in `idWrapper` is changed, the value in `id` is changed automatically:

```
ReadOnlyIntegerWrapper idWrapper = new ReadOnlyIntegerWrapper(100);
ReadOnlyIntegerProperty id = idWrapper.getReadOnlyProperty();

System.out.println("idWrapper:" + idWrapper.get());
System.out.println("id:" + id.get());

// Change the value
idWrapper.set(101);

System.out.println("idWrapper:" + idWrapper.get());
System.out.println("id:" + id.get());
```

```
idWrapper:100
id:100
idWrapper:101
id:101
```

■ **Tip** Typically, a wrapper property is used as a private instance variable of a class. The class can change the property internally. One of its methods returns the read-only property object of the wrapper class, so the same property is read-only for the outside world.

You can use seven types of properties that represent a single value. The base classes for those properties are named as `XXXProperty`, read-only base classes are named as `ReadOnlyXXXProperty`, and wrapper classes are named as `ReadOnlyXXXWrapper`. The values for `XXX` for each type are listed in Table 2-1.

Table 2-1. *List of Property Classes That Wrap a Single Value*

Type	XXX Value
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`String`	`String`
`Object`	`Object`

A property object wraps three pieces of information:

- The reference of the bean that contains it
- A name
- A value

When you create a property object, you can supply all or none of the preceding three pieces of information. Concrete property classes, named like `SimpleXXXProperty` and `ReadOnlyXXXWrapper`, provide four constructors that let you supply combinations of the three pieces of information. The following are the constructors for the `SimpleIntegerProperty` class:

```
SimpleIntegerProperty()
SimpleIntegerProperty(int initialValue)
SimpleIntegerProperty(Object bean, String name)
SimpleIntegerProperty(Object bean, String name, int initialValue)
```

The default value for the initial value depends on the type of the property. It is zero for numeric types, `false` for boolean types, and `null` for reference types.

A property object can be part of a bean, or it can be a stand-alone object. The specified `bean` is the reference to the bean object that contains the property. For a stand-alone property object, it can be `null`. Its default value is `null`.

The name of the property is its name. If not supplied, it defaults to an empty string.

The following snippet of code creates a property object as part of a bean and sets all three values. The first argument to the constructor of the `SimpleStringProperty` class is `this`, which is the reference of the `Person` bean, the second argument—`"name"`—is the name of the property, and the third argument—`"Li"`—is the value of the property:

```
public class Person {
        private StringProperty name = new SimpleStringProperty(
                this, "name", "Li");
           // More code goes here...
}
```

Every property class has `getBean()` and `getName()` methods that return the bean reference and the property name, respectively.

Using Properties in JavaFX Beans

In the previous section, you saw the use of JavaFX properties as stand-alone objects. In this section, you will use them in classes to define properties. Let's create a `Book` class with three properties: `ISBN`, `title`, and `price`, which will be modeled using JavaFX property classes.

In JavaFX, you do not declare the property of a class as one of the primitive types. Rather, you use one of the JavaFX property classes. The `title` property of the `Book` class will be declared as follows. It is declared `private` as usual:

```
public class Book {
        private StringProperty title = new SimpleStringProperty(this,
                "title", "Unknown");
}
```

You declare a public getter for the property, which is named, by convention, as `XXXProperty`, where `XXX` is the name of the property. This getter returns the reference of the property. For our `title` property, the getter will be named `titleProperty` as shown in the following:

```
public class Book {
        private StringProperty title = new SimpleStringProperty(this,
                "title", "Unknown");

        public final StringProperty titleProperty() {
                return title;
        }
}
```

The preceding declaration of the `Book` class is fine to work with the `title` property, as shown in the following snippet of code that sets and gets the title of a book:

```
Book b = new Book();
b.titleProperty().set("Harnessing JavaFX 17.0");
String title = b.titleProperty().get();
```

According to the JavaFX design patterns, and not for any technical requirements, a JavaFX property has a getter and a setter that are similar to the getters and setters in JavaBeans. The return type of the getter and the parameter type of the setter are the same as the type of the property value. For example, for `StringProperty` and `IntegerProperty`, they will be `String` and `int`, respectively. The `getTitle()` and `setTitle()` methods for the `title` property are declared as follows:

```
public class Book {
        private StringProperty title = new SimpleStringProperty(this,
                "title", "Unknown");

        public final StringProperty titleProperty() {
                return title;
        }

        public final String getTitle() {
                return title.get();
        }
```

```
        public final void setTitle(String title) {
                this.title.set(title);
        }
}
```

Note that the `getTitle()` and `setTitle()` methods use the `title` property object internally to get and set the title value.

■ **Tip** By convention, getters and setters for a property of a class are declared `final`. Additional getters and setters, using JavaBeans naming convention, are added to make the class interoperable with the older tools and frameworks that use the old JavaBeans naming conventions to identify the properties of a class.

The following snippet of code shows the declaration of a read-only `ISBN` property for the `Book` class:

```
public class Book {
        private ReadOnlyStringWrapper ISBN =
                new ReadOnlyStringWrapper(this, "ISBN", "Unknown");

        public final String getISBN() {
                return ISBN.get();
        }

        public final ReadOnlyStringProperty ISBNProperty() {
                return ISBN.getReadOnlyProperty();
        }

        // More code goes here...
}
```

Notice the following points about the declaration of the read-only `ISBN` property:

- It uses the `ReadOnlyStringWrapper` class instead of the `SimpleStringProperty` class.
- There is no setter for the property value. You may declare one; however, it must be private.
- The getter for the property value works the same as for a read/write property.
- The `ISBNProperty()` method uses `ReadOnlyStringProperty` as the return type, not `ReadOnlyStringWrapper`. It obtains a read-only version of the property object from the wrapper object and returns the same.

For the users of the `Book` class, its `ISBN` property is read-only. However, it can be changed internally, and the change will be reflected in the read-only version of the property object automatically.

Listing 2-3 shows the complete code for the `Book` class.

Listing 2-3. A Book Class with Two Read/Write and a Read-Only Properties

```
// Book.java
package com.jdojo.binding;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.ReadOnlyStringProperty;
import javafx.beans.property.ReadOnlyStringWrapper;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Book {
        private StringProperty title = new SimpleStringProperty(this,
               "title", "Unknown");
        private DoubleProperty price = new SimpleDoubleProperty(this,
                "price", 0.0);
        private ReadOnlyStringWrapper ISBN = new ReadOnlyStringWrapper(this,
                "ISBN", "Unknown");

        public Book() {
        }

        public Book(String title, double price, String ISBN) {
                this.title.set(title);
                this.price.set(price);
                this.ISBN.set(ISBN);
        }

        public final String getTitle() {
                return title.get();
        }

        public final void setTitle(String title) {
                this.title.set(title);
        }

        public final StringProperty titleProperty() {
                return title;
        }

        public final double getprice() {
                return price.get();
        }

        public final void setPrice(double price) {
                this.price.set(price);
        }

        public final DoubleProperty priceProperty() {
                return price;
        }
```

```
        public final String getISBN() {
                return ISBN.get();
        }

        public final ReadOnlyStringProperty ISBNProperty() {
                return ISBN.getReadOnlyProperty();
        }
}
```

Listing 2-4 tests the properties of the Book class. It creates a Book object, prints the details, changes some properties, and prints the details again. Note the use of the `ReadOnlyProperty` parameter type for the `printDetails()` method. All property classes implement, directly or indirectly, the `ReadOnlyProperty` interface.

The `toString()` methods of the property implementation classes return a well-formatted string that contains all relevant pieces of information for a property. I did not use the `toString()` method of the property objects because I wanted to show you the use of the different methods of the JavaFX properties.

Listing 2-4. A Test Class to Test Properties of the Book Class

```
// BookPropertyTest.java
package com.jdojo.binding;

import javafx.beans.property.ReadOnlyProperty;

public class BookPropertyTest {
        public static void main(String[] args) {
                Book book = new Book("Harnessing JavaFX", 9.99,
                          "0123456789");

                System.out.println("After creating the Book object...");

                // Print Property details
                printDetails(book.titleProperty());
                printDetails(book.priceProperty());
                printDetails(book.ISBNProperty());

                // Change the book's properties
                book.setTitle("Harnessing JavaFX 17.0");
                book.setPrice(9.49);

                System.out.println(
                         "\nAfter changing the Book properties...");

                // Print Property details
                printDetails(book.titleProperty());
                printDetails(book.priceProperty());
                printDetails(book.ISBNProperty());
        }
```

```
        public static void printDetails(ReadOnlyProperty<?> p) {
                String name = p.getName();
                Object value = p.getValue();
                Object bean = p.getBean();
                String beanClassName = (bean == null)?
                          "null":bean.getClass().getSimpleName();
                String propClassName = p.getClass().getSimpleName();

                System.out.print(propClassName);
                System.out.print("[Name:" + name);
                System.out.print(", Bean Class:" + beanClassName);
                System.out.println(", Value:" + value + "]");
        }
}
```

```
After creating the Book object...
SimpleStringProperty[Name:title, Bean Class:Book, Value:Harnessing JavaFX]
SimpleDoubleProperty[Name:price, Bean Class:Book, Value:9.99]
ReadOnlyPropertyImpl[Name:ISBN, Bean Class:Book, Value:0123456789]

After changing the Book properties...
SimpleStringProperty[Name:title, Bean Class:Book, Value:Harnessing JavaFX 17.0]
SimpleDoubleProperty[Name:price, Bean Class:Book, Value:9.49]
ReadOnlyPropertyImpl[Name:ISBN, Bean Class:Book, Value:0123456789]
```

Understanding the Property Class Hierarchy

It is important to understand a few core classes and interfaces of the JavaFX properties and binding APIs before you start using them. Figure 2-1 shows the class diagram for core interfaces of the properties API. You will not need to use these interfaces directly in your programs. Specialized versions of these interfaces and the classes that implement them exist and are used directly.

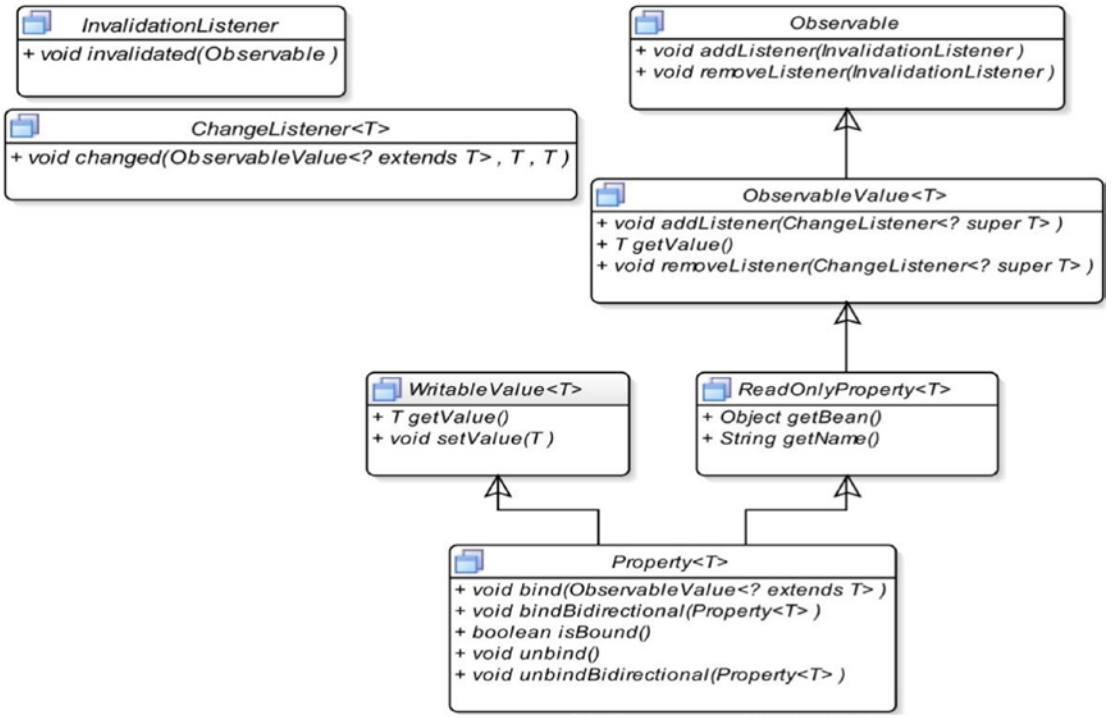

Figure 2-1. *A class diagram for core interfaces in the JavaFX property API*

Classes and interfaces in the JavaFX properties API are spread across different packages. Those packages are `javafx.beans`, `javafx.beans.binding`, `javafx.beans.property`, and `javafx.beans.value`.

The `Observable` interface is at the top of the properties API. An `Observable` wraps content, and it can be observed for invalidations of its content. The `Observable` interface has two methods to support this. Its `addListener()` method lets you add an `InvalidationListener`. The `invalidated()` method of the `InvalidationListener` is called when the content of the `Observable` becomes invalid. An `InvalidationListener` can be removed using its `removeListener()` method.

■ **Tip** All JavaFX properties are observable.

An `Observable` should generate an invalidation event only when the status of its content changes from valid to invalid. That is, multiple invalidations in a row should generate only one invalidation event. Property classes in the JavaFX follow this guideline.

■ **Tip** The generation of an invalidation event by an `Observable` does not necessarily mean that its content has changed. All it means is that its content is invalid for some reason. For example, sorting an `ObservableList` may generate an invalidation event. Sorting does not change the contents of the list; it only reorders the contents.

The ObservableValue interface inherits from the Observable interface. An ObservableValue wraps a value, which can be observed for changes. It has a getValue() method that returns the value it wraps. It generates invalidation events and change events. Invalidation events are generated when the value in the ObservableValue is no longer valid. Change events are generated when the value changes. You can register a ChangeListener to an ObservableValue. The changed() method of the ChangeListener is called every time the value of its value changes. The changed() method receives three arguments: the reference of the ObservableValue, the old value, and the new value.

An ObservableValue can recompute its value lazily or eagerly. In a lazy strategy, when its value becomes invalid, it does not know if the value has changed until the value is recomputed; the value is recomputed the next time it is read. For example, using the getValue() method of an ObservableValue would make it recompute its value if the value was invalid and if it uses a lazy strategy. In an eager strategy, the value is recomputed as soon as it becomes invalid.

To generate invalidation events, an ObservableValue can use lazy or eager evaluation. A lazy evaluation is more efficient. However, generating change events forces an ObservableValue to recompute its value immediately (an eager evaluation) as it has to pass the new value to the registered change listeners.

The ReadOnlyProperty interface adds getBean() and getName() methods. Their use was illustrated in Listing 2-4. The getBean() method returns the reference of the bean that contains the property object. The getName() method returns the name of the property. A read-only property implements this interface.

A WritableValue wraps a value that can be read and set using its getValue() and setValue() methods, respectively. A read/write property implements this interface.

The Property interface inherits from ReadOnlyProperty and WritableValue interfaces. It adds the following five methods to support binding:

- `void bind(ObservableValue<? extends T> observable)`
- `void unbind()`
- `void bindBidirectional(Property<T> other)`
- `void unbindBidirectional(Property<T> other)`
- `boolean isBound()`

The bind() method adds a unidirectional binding between this Property and the specified ObservableValue. The unbind() method removes the unidirectional binding for this Property, if one exists.

The bindBidirectional() method creates a bidirectional binding between this Property and the specified Property. The unbindBidirectional() method removes a bidirectional binding.

Note the difference in the parameter types for the bind() and bindBidirectional() methods. A unidirectional binding can be created between a Property and an ObservableValue of the same type as long as they are related through inheritance. However, a bidirectional binding can only be created between two properties of the same type.

The isBound() method returns true if the Property is bound. Otherwise, it returns false.

■ **Tip** All read/write JavaFX properties support binding.

Figure 2-2 shows a partial class diagram for the integer property in JavaFX. The diagram gives you an idea about the complexity of the JavaFX properties API. You do not need to learn all of the classes in the properties API. You will use only a few of them in your applications.

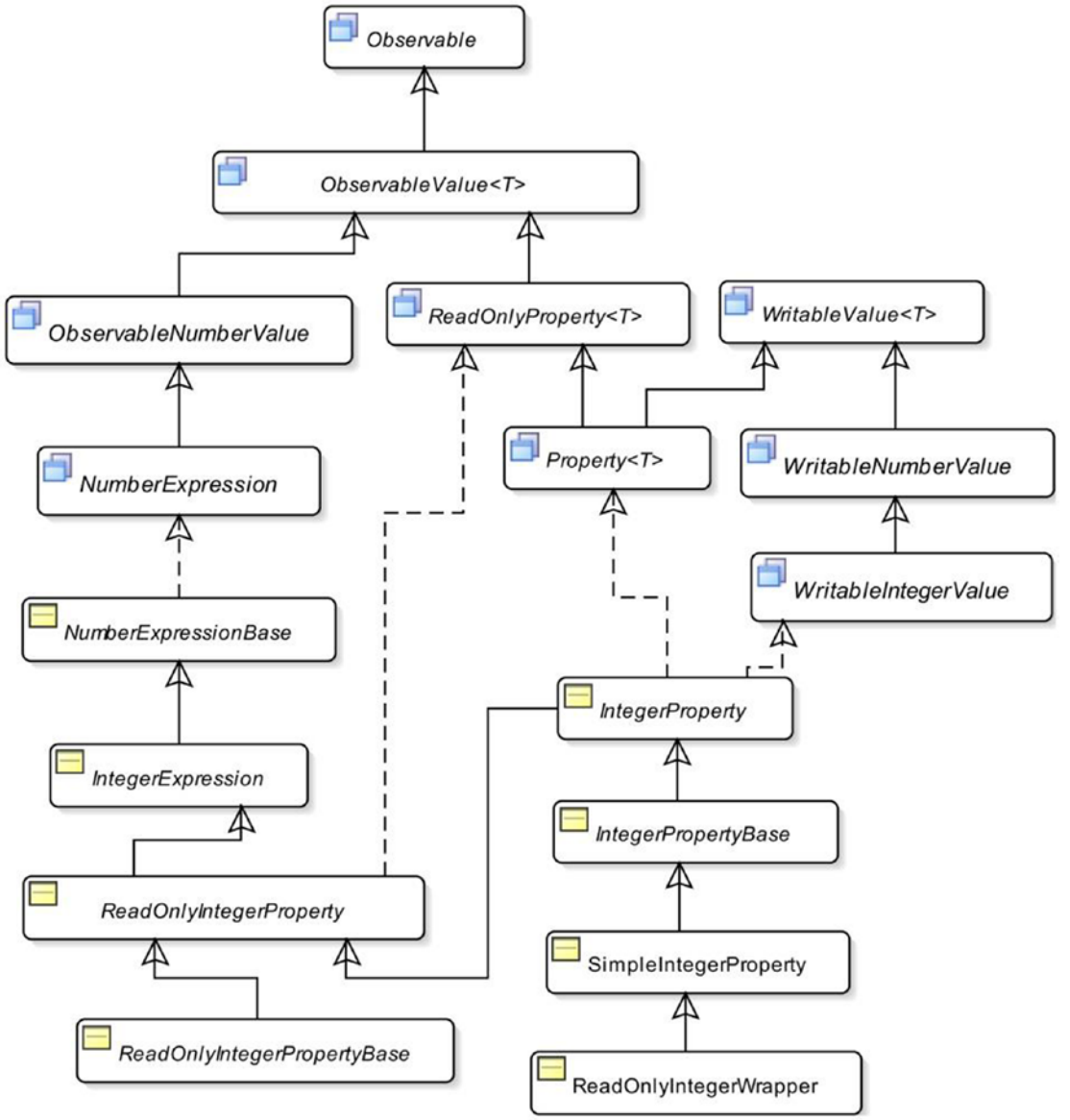

Figure 2-2. *A class diagram for the integer property*

Handling Property Invalidation Events

A property generates an invalidation event when the status of its value changes from valid to invalid for the first time. Properties in JavaFX use lazy evaluation. When an invalid property becomes invalid again, an invalidation event is not generated. An invalid property becomes valid when it is recomputed, for example, by calling its `get()` or `getValue()` method.

Listing 2-5 provides the program to demonstrate when invalidation events are generated for properties. The program includes enough comments to help you understand its logic. In the beginning, it creates an `IntegerProperty` named `counter`:

```
IntegerProperty counter = new SimpleIntegerProperty(100);
```

An `InvalidationListener` is added to the `counter` property:

```
counter.addListener(InvalidationTest::invalidated);
```

When you create a property object, it is valid. When you change the `counter` property to 101, it fires an invalidation event. At this point, the `counter` property becomes invalid. When you change its value to 102, it does not fire an invalidation event, because it is already invalid. When you use the `get()` method to read the `counter` value, it becomes valid again. Now you set the same value, 102, to the `counter`, which does not fire an invalidation event, as the value did not really change. The `counter` property is still valid. At the end, you change its value to a different value, and sure enough, an invalidation event is fired.

■ **Tip** You are not limited to adding only one invalidation listener to a property. You can add as many invalidation listeners as you need. Once you are done with an invalidation listener, make sure to remove it by calling the `removeListener()` method of the `Observable` interface; otherwise, it may lead to memory leaks.

Listing 2-5. Testing Invalidation Events for Properties

```
// InvalidationTest.java
// Listing part of the example sources download for the book
```

```
Before changing the counter value-1
Counter is invalid.
After changing the counter value-1

Before changing the counter value-2
After changing the counter value-2
Counter value = 102

Before changing the counter value-3
After changing the counter value-3

Before changing the counter value-4
Counter is invalid.
After changing the counter value-4
```

Handling Property Change Events

You can register a `ChangeListener` to receive notifications about property change events. A property change event is fired every time the value of a property changes. The `changed()` method of a `ChangeListener` receives three values: the reference of the property object, the old value, and the new value.

Let's run a similar test case for testing property change events as was done for invalidation events in the previous section. Listing 2-6 has the program to demonstrate change events that are generated for properties.

Listing 2-6. Testing Change Events for Properties

```
// ChangeTest.java
package com.jdojo.binding;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ObservableValue;

public class ChangeTest {
        public static void main(String[] args) {
            IntegerProperty counter = new SimpleIntegerProperty(100);

            // Add a change listener to the counter property
            counter.addListener(ChangeTest::changed);

            System.out.println("\nBefore changing the counter value-1");
            counter.set(101);
            System.out.println("After changing the counter value-1");

            System.out.println("\nBefore changing the counter value-2");
            counter.set(102);
            System.out.println("After changing the counter value-2");

            // Try to set the same value
            System.out.println("\nBefore changing the counter value-3");
            counter.set(102); // No change event is fired.
            System.out.println("After changing the counter value-3");

            // Try to set a different value
            System.out.println("\nBefore changing the counter value-4");
            counter.set(103);
            System.out.println("After changing the counter value-4");
        }

        public static void changed(ObservableValue<? extends Number> prop,
                                   Number oldValue,
                                   Number newValue) {
            System.out.print("Counter changed: ");
            System.out.println("Old = " + oldValue +
                    ", new = " + newValue);
        }
}
```

```
Before changing the counter value-1
Counter changed: Old = 100, new = 101
After changing the counter value-1

Before changing the counter value-2
Counter changed: Old = 101, new = 102
After changing the counter value-2

Before changing the counter value-3
After changing the counter value-3

Before changing the counter value-4
Counter changed: Old = 102, new = 103
After changing the counter value-4
```

In the beginning, the program creates an `IntegerProperty` named `counter`:

```
IntegerProperty counter = new SimpleIntegerProperty(100);
```

There is a little trick in adding a `ChangeListener`. The `addListener()` method in the `IntegerPropertyBase` class is declared as follows:

```
void addListener(ChangeListener<? super Number> listener)
```

This means that if you are using generics, the `ChangeListener` for an `IntegerProperty` must be written in terms of the `Number` class or a superclass of the `Number` class. Three ways to add a `ChangeListener` to the `counter` property are shown as follows:

```
// Method-1: Using generics and the Number class
counter.addListener(new ChangeListener<Number>() {
        @Override
        public void changed(ObservableValue<? extends Number> prop,
                            Number oldValue,
                            Number newValue) {
                System.out.print("Counter changed: ");
                System.out.println("Old = " + oldValue +
                        ", new = " + newValue);
        }});

// Method-2: Using generics and the Object class
counter.addListener( new ChangeListener<Object>() {
        @Override
        public void changed(ObservableValue<? extends Object> prop,
                            Object oldValue,
                            Object newValue) {
                System.out.print("Counter changed: ");
                System.out.println("Old = " + oldValue +
                        ", new = " + newValue);
        }});
```

```
// Method-3: Not using generics. It may generate compile-time warnings.
counter.addListener(new ChangeListener() {
        @Override
        public void changed(ObservableValue prop,
                            Object oldValue,
                            Object newValue) {
                System.out.print("Counter changed: ");
                System.out.println("Old = " + oldValue +
                         ", new = " + newValue);
        }});
```

Listing 2-6 uses the first method, which makes use of generics; as you can see, the signature of the `changed()` method in the `ChangeTest` class matches with the `changed()` method signature in `method-1`. I have used a lambda expression with a method reference to add a `ChangeListener` as shown:

```
counter.addListener(ChangeTest::changed);
```

The preceding output shows that a property change event is fired when the property value is changed. Calling the `set()` method with the same value does not fire a property change event.

Unlike generating invalidation events, a property uses an eager evaluation for its value to generate change events, because it has to pass the new value to the property change listeners. The next section discusses how a property object evaluates its value, if it has both invalidation and change listeners.

Handling Invalidation and Change Events

You need to consider performance when you have to decide between using invalidation listeners and change listeners. Generally, invalidation listeners perform better than change listeners. The reason is twofold:

- Invalidation listeners make it possible to compute the value lazily.
- Multiple invalidations in a row fire only one invalidation event.

However, which listener you use depends on the situation at hand. A rule of thumb is that if you read the value of the property inside the invalidation event handler, you should use a change listener instead. When you read the value of a property inside an invalidation listener, it triggers the recomputation of the value, which is automatically done before firing a change event. If you do not need to read the value of a property, use invalidation listeners.

Listing 2-7 has a program that adds an invalidation listener and a change listener to an `IntegerProperty`. This program is a combination of Listings 2-5 and 2-6. The output below it shows that when the property value changes, both events, invalidation and change, are always fired. This is because a change event makes a property valid immediately after the change, and the next change in the value fires an invalidation event and, of course, a change event too.

Listing 2-7. Testing Invalidation and Change Events for Properties Together

```
// ChangeAndInvalidationTest.java
// Listing part of the example sources download for the book
```

```
Before changing the counter value-1
Counter is invalid.
Counter changed: old = 100, new = 101
After changing the counter value-1

Before changing the counter value-2
Counter is invalid.
Counter changed: old = 101, new = 102
After changing the counter value-2

Before changing the counter value-3
After changing the counter value-3

Before changing the counter value-4
Counter is invalid.
Counter changed: old = 102, new = 103
After changing the counter value-4
```

Using Bindings in JavaFX

In JavaFX, a binding is an expression that evaluates to a value. It consists of one or more observable values known as its *dependencies*. A binding observes its dependencies for changes and recomputes its value automatically. JavaFX uses lazy evaluation for all bindings. When a binding is initially defined or when its dependencies change, its value is marked as invalid. The value of an invalid binding is computed when it is requested next time, usually using its `get()` or `getValue()` method. All property classes in JavaFX have built-in support for binding.

Let's look at a quick example of binding in JavaFX. Consider the following expression that represents the sum of two integers x and y:

```
x + y
```

The expression, x + y, represents a binding, which has two dependencies: x and y. You can give it a name `sum` as

```
sum = x + y
```

To implement the preceding logic in JavaFX, you create two `IntegerProperty` variables: `x` and `y`:

```
IntegerProperty x = new SimpleIntegerProperty(100);
IntegerProperty y = new SimpleIntegerProperty(200);
```

The following statement creates a binding named `sum` that represents the sum of `x` and `y`:

```
NumberBinding sum = x.add(y);
```

A binding has an `isValid()` method that returns `true` if it is valid; otherwise, it returns `false`. You can get the value of a `NumberBinding` using the methods `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()` as `int`, `long`, `float`, and `double`, respectively.

The program in Listing 2-8 shows how to create and use a binding based on the preceding discussion. When the sum binding is created, it is invalid and it does not know its value. This is evident from the output. Once you request its value, using the `sum.initValue()` method, it computes its value and marks itself as valid. When you change one of its dependencies, it becomes invalid until you request its value again.

Listing 2-8. Using a Simple Binding

```
// BindingTest.java
package com.jdojo.binding;

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

public class BindingTest {
        public static void main(String[] args) {
            IntegerProperty x = new SimpleIntegerProperty(100);
            IntegerProperty y = new SimpleIntegerProperty(200);

            // Create a binding: sum = x + y
            NumberBinding sum = x.add(y);

            System.out.println("After creating sum");
            System.out.println("sum.isValid(): " + sum.isValid());

            // Let us get the value of sum, so it computes its value and
            // becomes valid
            int value = sum.intValue();

            System.out.println("\nAfter requesting value");
            System.out.println("sum.isValid(): " + sum.isValid());
            System.out.println("sum = " + value);

            // Change the value of x
            x.set(250);

            System.out.println("\nAfter changing x");
            System.out.println("sum.isValid(): " + sum.isValid());

            // Get the value of sum again
            value = sum.intValue();

            System.out.println("\nAfter requesting value");
            System.out.println("sum.isValid(): " + sum.isValid());
            System.out.println("sum = " + value);
        }
}
```

```
After creating sum
sum.isValid(): false

After requesting value
sum.isValid(): true
sum = 300

After changing x
sum.isValid(): false

After requesting value
sum.isValid(): true
sum = 450
```

A binding, internally, adds invalidation listeners to all of its dependencies (Listing 2-9). When any of its dependencies become invalid, it marks itself as invalid. An invalid binding does not mean that its value has changed. All it means is that it needs to recompute its value when the value is requested next time.

In JavaFX, you can also bind a property to a binding. Recall that a binding is an expression that is synchronized with its dependencies automatically. Using this definition, a bound property is a property whose value is computed based on an expression, which is automatically synchronized when the dependencies change. Suppose you have three properties, x, y, and z, as follows:

```
IntegerProperty x = new SimpleIntegerProperty(10);
IntegerProperty y = new SimpleIntegerProperty(20);
IntegerProperty z = new SimpleIntegerProperty(60);
```

You can bind the property z to an expression, `x + y`, using the `bind()` method of the `Property` interface as follows:

```
z.bind(x.add(y));
```

Note that you cannot write `z.bind(x + y)` as the + operator does not know how to add the values of two `IntegerProperty` objects. You need to use the binding API, as you did in the preceding statement, to create a binding expression. I will cover the details of the binding API shortly.

Now, when x, y, or both change, the z property becomes invalid. The next time you request the value of z, it recomputes the expression `x.add(y)` to get its value.

You can use the `unbind()` method of the `Property` interface to unbind a bound property. Calling the `unbind()` method on an unbound or never bound property has no effect. You can unbind the z property as follows:

```
z.unbind();
```

After unbinding, a property behaves as a normal property, maintaining its value independently. Unbinding a property breaks the link between the property and its dependencies.

Listing 2-9. Binding a Property

```
// BoundProperty.java
package com.jdojo.binding;

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

public class BoundProperty {
        public static void main(String[] args) {
            IntegerProperty x = new SimpleIntegerProperty(10);
            IntegerProperty y = new SimpleIntegerProperty(20);
            IntegerProperty z = new SimpleIntegerProperty(60);
            z.bind(x.add(y));
            System.out.println("After binding z: Bound = " + z.isBound() +
                ", z = " + z.get());

            // Change x and y
            x.set(15);
            y.set(19);
            System.out.println("After changing x and y: Bound = " +
                    z.isBound() + ", z = " + z.get());
            // Unbind z
            z.unbind();

            // Will not affect the value of z as it is not bound to
                // x and y anymore
            x.set(100);
            y.set(200);
            System.out.println("After unbinding z: Bound = " +
                    z.isBound() + ", z = " + z.get());
        }
}
```

```
After binding z: Bound = true, z = 30
After changing x and y: Bound = true, z = 34
After unbinding z: Bound = false, z = 34
```

Unidirectional and Bidirectional Bindings

A binding has a direction, which is the direction in which changes are propagated. JavaFX supports two types of binding for properties: *unidirectional binding* and *bidirectional binding*. A unidirectional binding works only in one direction; changes in dependencies are propagated to the bound property and not vice versa. A bidirectional binding works in both directions; changes in dependencies are reflected in the property and vice versa.

The `bind()` method of the `Property` interface creates a unidirectional binding between a property and an `ObservableValue`, which could be a complex expression. The `bindBidirectional()` method creates a bidirectional binding between a property and another property of the same type.

Suppose that x, y, and z are three instances of `IntegerProperty`. Consider the following bindings:

```
z = x + y
```

In JavaFX, the preceding binding can only be expressed as a unidirectional binding as follows:

```
z.bind(x.add(y));
```

Suppose you were able to use bidirectional binding in the preceding case. If you were able to change the value of z to 100, how would you compute the values of x and y in the reverse direction? For z being 100, there are an infinite number of possible combinations for x and y, for example, (99, 1), (98, 2), (101, –1), (200, –100), and so on. Propagating changes from a bound property to its dependencies is not possible with predictable results. This is the reason that binding a property to an expression is allowed only as a unidirectional binding.

Unidirectional binding has a restriction. Once a property has a unidirectional binding, you cannot change the value of the property directly; its value must be computed automatically based on the binding. You must unbind it before changing its value directly. The following snippet of code shows this case:

```
IntegerProperty x = new SimpleIntegerProperty(10);
IntegerProperty y = new SimpleIntegerProperty(20);
IntegerProperty z = new SimpleIntegerProperty(60);
z.bind(x.add(y));

z.set(7878); // Will throw a RuntimeException
```

To change the value of z directly, you can type the following:

```
z.unbind();  // Unbind z first
z.set(7878); // OK
```

Unidirectional binding has another restriction. A property can have only one unidirectional binding at a time. Consider the following two unidirectional bindings for a property z. Assume that x, y, z, a, and b are five instances of `IntegerProperty`:

```
z = x + y
z = a + b
```

If x, y, a, and b are four different properties, the bindings shown earlier for z are not possible. Think about `x = 1`, `y = 2`, `a = 3`, and `b = 4`. Can you define the value of z? Will it be 3 or 7? This is the reason that a property can have only one unidirectional binding at a time.

Rebinding a property that already has a unidirectional binding unbinds the previous binding. For example, the following snippet of code works fine:

```
IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);
IntegerProperty a = new SimpleIntegerProperty(3);
IntegerProperty b = new SimpleIntegerProperty(4);
IntegerProperty z = new SimpleIntegerProperty(0);

z.bind(x.add(y));
System.out.println("z = " + z.get());
```

```
z.bind(a.add(b)); // Will unbind the previous binding
System.out.println("z = " + z.get());
```

```
z = 3
z = 7
```

A bidirectional binding works in both directions. It has some restrictions. It can only be created between properties of the same type. That is, a bidirectional binding can only be of the type `x = y` and `y = x`, where `x` and `y` are of the same type.

Bidirectional binding removes some restrictions that are present for unidirectional binding. A property can have multiple bidirectional bindings at the same time. A bidirectional bound property can also be changed independently; the change is reflected in all properties that are bound to this property. That is, the following bindings are possible, using the bidirectional bindings:

```
x = y
x = z
```

In the preceding case, the values of `x`, `y`, and `z` will always be synchronized. That is, all three properties will have the same value, after the bindings are established. You can also establish bidirectional bindings between `x`, `y`, and `z` as follows:

```
x = z
z = y
```

Now a question arises. Will both of the preceding bidirectional bindings end up having the same values in `x`, `y`, and `z`? The answer is no. The value of the right-hand operand (see the preceding expressions, for example) in the last bidirectional binding is the value that is contained by all participating properties. Let me elaborate this point. Suppose `x` is 1, `y` is 2, and `z` is 3, and you have the following bidirectional bindings:

```
x = y
x = z
```

The first binding, `x = y`, will set the value of `x` equal to the value of `y`. At this point, `x` and `y` will be 2. The second binding, `x = z`, will set the value of `x` to be equal to the value of `z`. That is, `x` and `z` will be 3. However, `x` already has a bidirectional binding to `y`, which will propagate the new value 3 of `x` to `y` as well. Therefore, all three properties will have the same value as that of `z`. The program in Listing 2-10 shows how to use bidirectional bindings.

Listing 2-10. Using Bidirectional Bindings

```
// BidirectionalBinding.java
package com.jdojo.binding;

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

public class BidirectionalBinding {
        public static void main(String[] args) {
            IntegerProperty x = new SimpleIntegerProperty(1);
            IntegerProperty y = new SimpleIntegerProperty(2);
            IntegerProperty z = new SimpleIntegerProperty(3);
```

```
            System.out.println("Before binding:");
            System.out.println("x=" + x.get() + ", y=" + y.get() +
                    ", z=" + z.get());

            x.bindBidirectional(y);
            System.out.println("After binding-1:");
            System.out.println("x=" + x.get() + ", y=" + y.get() +
                    ", z=" + z.get());

            x.bindBidirectional(z);
            System.out.println("After binding-2:");
            System.out.println("x=" + x.get() + ", y=" + y.get() +
                    ", z=" + z.get());

            System.out.println("After changing z:");
            z.set(19);
            System.out.println("x=" + x.get() + ", y=" + y.get() +
                    ", z=" + z.get());

            // Remove bindings
            x.unbindBidirectional(y);
            x.unbindBidirectional(z);
            System.out.println(
                    "After unbinding and changing them separately:");
            x.set(100);
            y.set(200);
            z.set(300);
            System.out.println("x=" + x.get() + ", y=" + y.get() +
                    ", z=" + z.get());
        }
}
```

```
Before binding:
x=1, y=2, z=3
After binding-1:
x=2, y=2, z=3
After binding-2:
x=3, y=3, z=3
After changing z:
x=19, y=19, z=19
After unbinding and changing them separately:
x=100, y=200, z=300
```

Unlike a unidirectional binding, when you create a bidirectional binding, the previous bindings are not removed because a property can have multiple bidirectional bindings. You must remove all bidirectional bindings using the `unbindBidirectional()` method, calling it once for each bidirectional binding for a property, as shown here:

```
// Create bidirectional bindings
x.bindBidirectional(y);
x.bindBidirectional(z);

// Remove bidirectional bindings
x.unbindBidirectional(y);
x.unbindBidirectional(z);
```

Understanding the Binding API

Previous sections gave you a quick and simple introduction to bindings in JavaFX. Now it's time to dig deeper and understand the binding API in detail. The binding API is divided into two categories:

- High-level binding API
- Low-level binding API

The high-level binding API lets you define binding using the JavaFX class library. For most use cases, you can use the high-level binding API.

Sometimes, the existing API is not sufficient to define a binding. In those cases, the low-level binding API is used. In the low-level binding API, you derive a binding class from an existing binding class and write your own logic to define a binding.

The High-Level Binding API

The high-level binding API consists of two parts: the Fluent API and the `Bindings` class. You can define bindings using only the Fluent API, only the `Bindings` class, or by combining the two. Let's look at both parts, first separately and then together.

Using the Fluent API

The Fluent API consists of several methods in different interfaces and classes. The API is called *Fluent* because the method names, their parameters, and return types have been designed in such a way that they allow writing the code fluently. The code written using the Fluent API is more readable as compared to code written using nonfluent APIs. Designing a fluent API takes more time. A fluent API is more developer friendly and less designer friendly. One of the features of a fluent API is *method chaining*; you can combine separate method calls into one statement. Consider the following snippet of code to add three properties `x`, `y`, and `z`. The code using a nonfluent API might look as follows:

```
x.add(y);
x.add(z);
```

Using a Fluent API, the preceding code may look as shown in the following, which gives readers a better understanding of the intention of the writer:

```
x.add(y).add(z);
```

Figure 2-3 shows a class diagram for the `IntegerBinding` and `IntegerProperty` classes. The diagram has omitted some of the interfaces and classes that fall into the `IntegerProperty` class hierarchy. Class diagrams for `long`, `float`, and `double` types are similar.

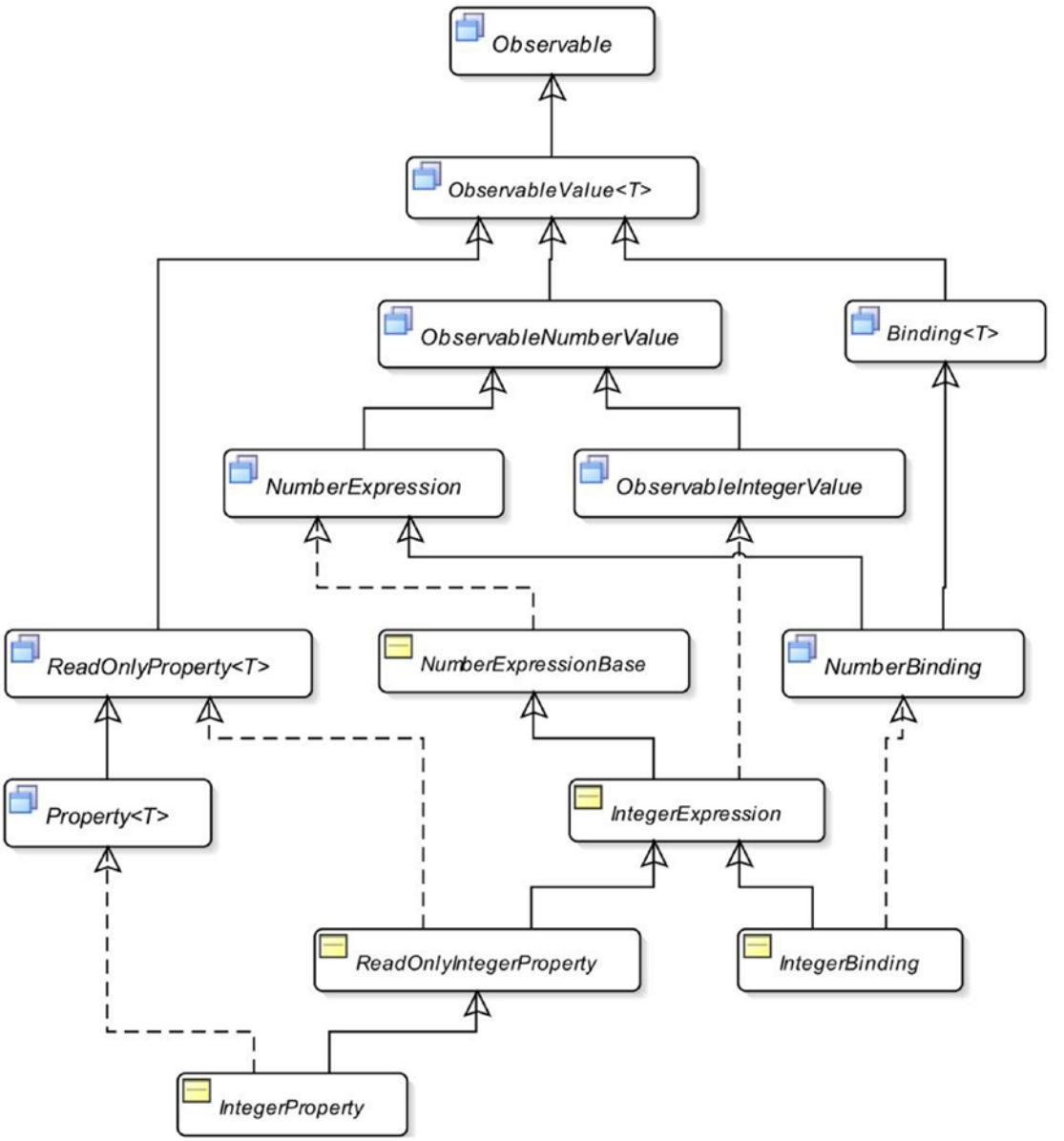

Figure 2-3. *A partial class diagram for* `IntegerBinding` *and* `IntegerProperty`

Classes and interfaces from the `ObservableNumberValue` and `Binding` interfaces down to the `IntegerBinding` class are part of the fluent binding API for the `int` data type. At first, it may seem as if there were many classes to learn. Most of the classes and interfaces exist in properties and binding APIs to avoid boxing and unboxing of primitive values. To learn the fluent binding API, you need to focus on `XXXExpression` and `XXXBinding` classes and interfaces. The `XXXExpression` classes have the methods that are used to create binding expressions.

The *Binding* Interface

An instance of the `Binding` interface represents a value that is derived from one or more sources known as dependencies. It has the following four methods:

- `public void dispose()`
- `public ObservableList<?> getDependencies()`
- `public void invalidate()`
- `public boolean isValid()`

The `dispose()` method, whose implementation is optional, indicates to a `Binding` that it will no longer be used, so it can remove references to other objects. The binding API uses weak invalidation listeners internally, making the call to this method unnecessary.

The `getDependencies()` method, whose implementation is optional, returns an unmodifiable `ObservableList` of dependencies. It exists only for debugging purposes. This method should not be used in production code.

A call to the `invalidate()` method invalidates a `Binding`. The `isValid()` method returns `true` if a `Binding` is valid. Otherwise, it returns `false`.

The *NumberBinding* Interface

The `NumberBinding` interface is a marker interface whose instance wraps a numeric value of `int`, `long`, `float`, or `double` type. It is implemented by `DoubleBinding`, `FloatBinding`, `IntegerBinding`, and `LongBinding` classes.

The *ObservableNumberValue* Interface

An instance of the `ObservableNumberValue` interface wraps a numeric value of `int`, `long`, `float`, or `double` type. It provides the following four methods to get the value:

- `double doubleValue()`
- `float floatValue()`
- `int intValue()`
- `long longValue()`

You used the `intValue()` method provided in Listing 2-8 to get the `int` value from a `NumberBinding` instance. The code you use would be

```
IntegerProperty x = new SimpleIntegerProperty(100);
IntegerProperty y = new SimpleIntegerProperty(200);

// Create a binding: sum = x + y
NumberBinding sum = x.add(y);
int value = sum.intValue(); // Get the int value
```

The *ObservableIntegerValue* Interface

The `ObservableIntegerValue` interface defines a `get()` method that returns the type-specific `int` value.

The *NumberExpression* Interface

The `NumberExpression` interface contains several convenience methods to create bindings using a fluent style. It has over 50 methods, and most of them are overloaded. These methods return a `Binding` type such as `NumberBinding`, `BooleanBinding`, and so on. Table 2-2 lists the methods in the `NumberExpression` interface. Most of the methods are overloaded. The table does not show the method arguments.

Table 2-2. *Summary of the Methods in the `NumberExpression` Interface*

Method Name	Return Type	Description

`add()`
`subtract()`
`multiply()`
`divide()`
	`NumberBinding`	These methods create a new `NumberBinding` that is the sum, difference, product, and division of the `NumberExpression`, and a numeric value or an `ObservableNumberValue`.

`greaterThan()`
`greaterThanOrEqualTo()`
`isEqualTo()`
`isNotEqualTo()`
`lessThan()`
`lessThanOrEqualTo()`
	`BooleanBinding`	These methods create a new `BooleanBinding` that stores the result of the comparison of the `NumberExpression` and a numeric value or an `ObservableNumberValue`. Method names are clear enough to tell what kind of comparisons they perform.

`negate()`	`NumberBinding`	It creates a new `NumberBinding` that is the negation of the `NumberExpression`.
`asString()`	`StringBinding`	It creates a `StringBinding` that holds the value of the `NumberExpression` as a `String` object. This method also supports locale-based string formatting.

The methods in the `NumberExpression` interface allow for mixing types (`int`, `long`, `float`, and `double`) while defining a binding, using an arithmetic expression. When the return type of a method in this interface is `NumberBinding`, the actual returned type would be of `IntegerBinding`, `LongBinding`, `FloatBinding`, or `DoubleBinding`. The binding type of an arithmetic expression is determined by the same rules as the Java programming language. The results of an expression depend on the types of the operands. The rules are as follows:

- If one of the operands is a `double`, the result is a `double`.
- If none of the operands is a `double` and one of them is a `float`, the result is a `float`.
- If none of the operands is a `double` or a `float` and one of them is a `long`, the result is a `long`.
- Otherwise, the result is an `int`.

Consider the following snippet of code:

```
IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);
NumberBinding sum = x.add(y);
int value = sum.intValue();
```

The number expression `x.add(y)` involves only `int` operands (`x` and `y` are of `int` type). Therefore, according to the preceding rules, its result is an `int` value, and it returns an `IntegerBinding` object. Because the `add()` method in the `NumberExpression` specifies the return type as `NumberBinding`, a `NumberBinding` type is used to store the result. You have to use the `intValue()` method from the `ObservableNumberValue` interface. You can rewrite the preceding snippet of code as follows:

```
IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);

// Casting to IntegerBinding is safe
IntegerBinding sum = (IntegerBinding)x.add(y);
int value = sum.get();
```

The `NumberExpressionBase` class is an implementation of the `NumberExpression` interface. The `IntegerExpression` class extends the `NumberExpressionBase` class. It overrides methods in its superclass to provide a type-specific return type.

The program in Listing 2-11 creates a `DoubleBinding` that computes the area of a circle. It also creates a `DoubleProperty` and binds it to the same expression to compute the area. It is your choice whether you want to work with `Binding` objects or bound property objects. The program shows you both approaches.

Listing 2-11. Computing the Area of a Circle from Its Radius Using a Fluent Binding API

```
// CircleArea.java
package com.jdojo.binding;

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

public class CircleArea {
        public static void main(String[] args) {
            DoubleProperty radius = new SimpleDoubleProperty(7.0);

            // Create a binding for computing area of the circle
            DoubleBinding area =
                    radius.multiply(radius).multiply(Math.PI);

            System.out.println("Radius = " + radius.get() +
                        ", Area = " + area.get());

            // Change the radius
            radius.set(14.0);
            System.out.println("Radius = " + radius.get() +
                    ", Area = " + area.get());

            // Create a DoubleProperty and bind it to an expression
            // that computes the area of the circle
            DoubleProperty area2 = new SimpleDoubleProperty();
            area2.bind(radius.multiply(radius).multiply(Math.PI));
            System.out.println("Radius = " + radius.get() +
                        ", Area2 = " + area2.get());
        }
}
```

```
Radius = 7.0, Area = 153.93804002589985
Radius = 14.0, Area = 615.7521601035994
Radius = 14.0, Area2 = 615.7521601035994
```

The *StringBinding* Class

The class diagram containing classes in the binding API that supports binding of `String` type is depicted in Figure 2-4.

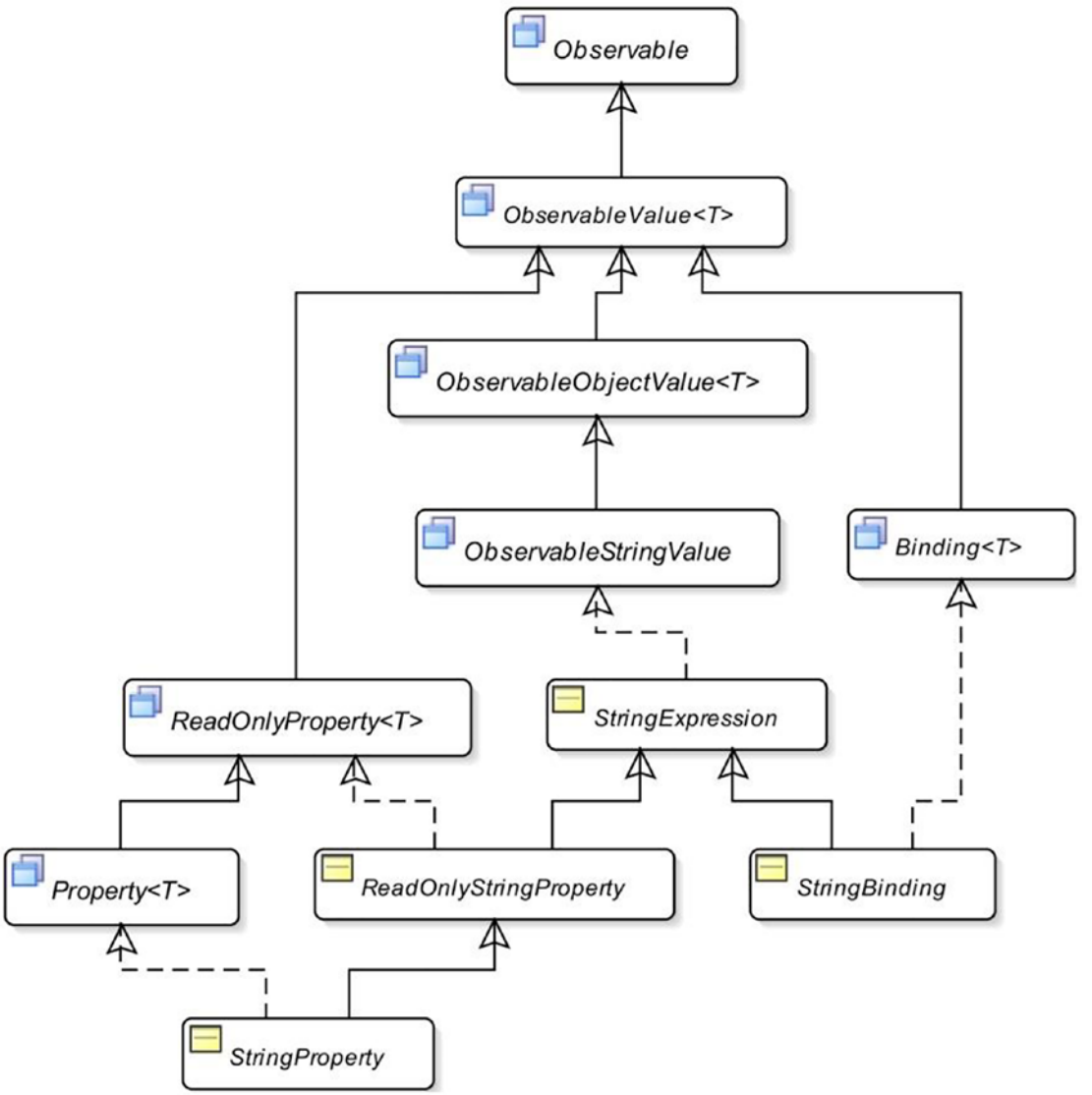

Figure 2-4. *A partial class diagram for* `StringBinding`

The ObservableStringValue interface declares a get() method whose return type is String. The methods in the StringExpression class let you create binding using a fluent style. Methods are provided to concatenate an object to the StringExpression, compare two strings, and check for null, among others. It has two methods to get its value: getValue() and getValueSafe(). Both return the current value. However, the latter returns an empty String when the current value is null.

The program in Listing 2-12 shows how to use StringBinding and StringExpression classes. The concat() method in the StringExpression class takes an Object type as an argument. If the argument is an ObservableValue, the StringExpression is updated automatically when the argument changes. Note the use of the asString() method on the radius and area properties. The asString() method on a NumberExpression returns a StringBinding.

Listing 2-12. Using StringBinding and StringExpression

```
// StringExpressionTest.java
package com.jdojo.binding;

import java.util.Locale;
import javafx.beans.binding.StringExpression;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class StringExpressionTest {
        public static void main(String[] args) {
            DoubleProperty radius = new SimpleDoubleProperty(7.0);
            DoubleProperty area = new SimpleDoubleProperty(0);
            StringProperty initStr = new SimpleStringProperty(
                    "Radius = ");

            // Bind area to an expression that computes the area of
            // the circle
            area.bind(radius.multiply(radius).multiply(Math.PI));

            // Create a string expression to describe the circle
            StringExpression desc = initStr.concat(radius.asString())
                .concat(", Area = ")
               .concat(area.asString(Locale.US, "%.2f"));

            System.out.println(desc.getValue());

            // Change the radius
            radius.set(14.0);
            System.out.println(desc.getValue());
        }
}
```

```
Radius = 7.0, Area = 153.94
Radius = 14.0, Area = 615.75
```

The *ObjectExpression* and *ObjectBinding* Classes

Now it's time for `ObjectExpression` and `ObjectBinding` classes to create bindings of any type of objects. Their class diagram is very similar to that of the `StringExpression` and `StringBinding` classes. The `ObjectExpression` class has methods to compare objects for equality and to check for null values. The program in Listing 2-13 shows how to use the `ObjectBinding` class.

Listing 2-13. Using the `ObjectBinding` Class

```
// ObjectBindingTest.java
package com.jdojo.binding;

import javafx.beans.binding.BooleanBinding;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;

public class ObjectBindingTest {
        public static void main(String[] args) {
            Book b1 = new Book("J1", 90, "1234567890");
            Book b2 = new Book("J2", 80, "0123456789");
            ObjectProperty<Book> book1 = new SimpleObjectProperty<>(b1);
            ObjectProperty<Book> book2 = new SimpleObjectProperty<>(b2);

            // Create a binding that computes if book1 and book2 are equal
            BooleanBinding isEqual = book1.isEqualTo(book2);
            System.out.println(isEqual.get());

            book2.set(b1);
            System.out.println(isEqual.get());
        }
}
```

```
false
true
```

The *BooleanExpression* and *BooleanBinding* Classes

The `BooleanExpression` class contains methods such as `and()`, `or()`, and `not()` that let you use boolean logical operators in an expression. Its `isEqualTo()` and `isNotEqualTo()` methods let you compare a `BooleanExpression` with another `ObservableBooleanValue`. The result of a `BooleanExpression` is `true` or `false`.

The program in Listing 2-14 shows how to use the `BooleanExpression` class. It creates a boolean expression, `x > y && y <> z`, using a fluent style. Note that the `greaterThan()` and `isNotEqualTo()` methods are defined in the `NumberExpression` interface. The program only uses the `and()` method from the `BooleanExpression` class.

Listing 2-14. Using `BooleanExpression` and `BooleanBinding`

```
// BooelanExpressionTest.java
package com.jdojo.binding;

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

public class BooelanExpressionTest {
        public static void main(String[] args) {
            IntegerProperty x = new SimpleIntegerProperty(1);
            IntegerProperty y = new SimpleIntegerProperty(2);
            IntegerProperty z = new SimpleIntegerProperty(3);

            // Create a boolean expression for x > y && y <> z
            BooleanExpression condition =
                    x.greaterThan(y).and(y.isNotEqualTo(z));

            System.out.println(condition.get());

            // Make the condition true by setting x to 3
            x.set(3);
            System.out.println(condition.get());
        }
}
```

```
false
true
```

Using Ternary Operation in Expressions

The Java programming language offers a ternary operator, (`condition?value1:value2`), to perform a ternary operation of the form *when-then-otherwise*. The JavaFX binding API has a `When` class for this purpose. The general syntax of using the `When` class is shown here:

```
new When(condition).then(value1).otherwise(value2)
```

The `condition` must be an `ObservableBooleanValue`. When the `condition` evaluates to `true`, it returns `value1`. Otherwise, it returns `value2`. The types of `value1` and `value2` must be the same. Values may be constants or instances of `ObservableValue`.

Let's use a ternary operation that returns a `String` `even` or `odd` depending on whether the value of an `IntegerProperty` is even or odd, respectively. The Fluent API does not have a method to compute modulus. You will have to do this yourself. Perform an integer division by 2 on an integer and multiply the result by 2. If you get the same number back, the number is even. Otherwise, the number is odd. For example, using an integer division, (7/2)*2, results in 6, and not 7. Listing 2-15 provides the complete program.

Listing 2-15. Using the `When` Class to Perform a Ternary Operation

```
// TernaryTest.java
package com.jdojo.binding;

import javafx.beans.binding.When;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.binding.StringBinding;

public class TernaryTest {
        public static void main(String[] args) {
            IntegerProperty num = new SimpleIntegerProperty(10);
            StringBinding desc =
                    new When(num.divide(2).multiply(2).isEqualTo(num))
                                        .then("even")
                                        .otherwise("odd");

            System.out.println(num.get() + " is " + desc.get());

            num.set(19);
            System.out.println(num.get() + " is " + desc.get());
        }
}
```

```
10 is even
19 is odd
```

Using the *Bindings* Utility Class

The `Bindings` class is a helper class to create simple bindings. It consists of more than 150 static methods. Most of them are overloaded with several variants. I will not list or discuss all of them. Please refer to the online JavaFX API documentation to get the complete list of methods. Table 2-3 lists the methods of the `Bindings` class and their descriptions. It has excluded methods belonging to collections binding.

Table 2-3. *Summary of Methods in the* `Bindings` *Class*

Method Name	Description

`add()`
`subtract()`
`multiply()`
`divide()`
	They create a binding by applying an arithmetic operation, indicated by their names, on two of its arguments. At least one of the arguments must be an `ObservableNumberValue`. If one of the arguments is a `double`, its return type is `DoubleBinding`; otherwise, its return type is `NumberBinding`.

`and()`	It creates a `BooleanBinding` by applying the boolean `and` to two of its arguments.
`bindBidirectional()` `unbindBidirectional()`	They create and delete a bidirectional binding between two properties.
`concat()`	It returns a `StringExpression` that holds the value of the concatenation of its arguments. It takes a `varargs` argument.
`convert()`	It returns a `StringExpression` that wraps its argument.
`createXXXBinding()`	It lets you create a custom binding of `XXX` type, where `XXX` could be `Boolean`, `Double`, `Float`, `Integer`, `String`, and `Object`.

`equal()`
`notEqual()`
`equalIgnoreCase()`
`notEqualIgnoreCase()`
	They create a `BooleanBinding` that wraps the result of comparing two of its arguments being equal or not equal. Some variants of the methods allow passing a tolerance value. If two arguments are within the tolerance, they are considered equal. Generally, a tolerance value is used to compare floating-point numbers. The ignore case variants of the methods work only on `String` type.

`format()`	It creates a `StringExpression` that holds the value of multiple objects formatted according to a specified format `String`.

`greaterThan()`
`greaterThanOrEqual()`
`lessThan()`
`lessThanOrEqual()`
	They create a `BooleanBinding` that wraps the result of comparing arguments.

`isNotNull` `isNull`	They create a `BooleanBinding` that wraps the result of comparing the argument with `null`.
`max()` `min()`	They create a binding that holds the maximum and minimum of two arguments of the method. One of the arguments must be an `ObservableNumberValue`.
`negate()`	It creates a `NumberBinding` that holds the negation of an `ObservableNumberValue`.
`not()`	It creates a `BooleanBinding` that holds the inverse of an `ObservableBooleanValue`.
`or()`	It creates a `BooleanBinding` that holds the result of applying the conditional `or` operation on its two `ObservableBooleanValue` arguments.
`selectXXX()`	It creates a binding to select a nested property. The nested property may be of the type `a.b.c`. The value of the binding will be `c`. The classes and properties involved in the expression like `a.b.c` must be public. If any part of the expression is not accessible, because they are not public or they do not exist, the default value for the type, for example, `null` for `Object type`, an empty `String` for `String type`, 0 for numeric type, and `false` for boolean type, is the value of the binding. (Later, I will discuss an example of using the `select()` method.)
`when()`	It creates an instance of the `When` class taking a condition as an argument.

Most of our examples using the Fluent API can also be written using the `Bindings` class. The program in Listing 2-16 is similar to the one in Listing 2-12. It uses the `Bindings` class instead of the Fluent API. It uses the `multiply()` method to compute the area and the `format()` method to format the results. There may be several ways of doing the same thing. For formatting the result, you can also use the `Bindings.concat()` method, as shown here:

```
StringExpression desc = Bindings.concat("Radius = ",
    radius.asString(Locale.US, "%.2f"),
   ", Area = ", area.asString(Locale.US, "%.2f"));
```

Listing 2-16. Using the `Bindings` Class

```
// BindingsClassTest.java
package com.jdojo.binding;

import java.util.Locale;
import javafx.beans.binding.Bindings;
import javafx.beans.binding.StringExpression;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class BindingsClassTest {
        public static void main(String[] args) {
            DoubleProperty radius = new SimpleDoubleProperty(7.0);
            DoubleProperty area = new SimpleDoubleProperty(0.0);

            // Bind area to an expression that computes the area of
            // the circle
            area.bind(Bindings.multiply(
                    Bindings.multiply(radius, radius), Math.PI));

            // Create a string expression to describe the circle
            StringExpression desc = Bindings.format(Locale.US,
                "Radius = %.2f, Area = %.2f", radius, area);

            System.out.println(desc.get());

            // Change the radius
            radius.set(14.0);
            System.out.println(desc.getValue());
        }
}
```

```
Radius = 7.00, Area = 153.94
Radius = 14.00, Area = 615.75
```

Let's look at an example of using the `selectXXX()` method of the `Bindings` class. It is used to create a binding for a nested property. In the nested hierarchy, all classes and properties must be public. Suppose you have an `Address` class that has a `zip` property and a `Person` class that has an `addr` property. The classes are shown in Listings 2-17 and 2-18, respectively.

Listing 2-17. An Address Class

```
// Address.java
package com.jdojo.binding;

import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Address {
        private StringProperty zip = new SimpleStringProperty("36106");

        public StringProperty zipProperty() {
                return zip;
        }
}
```

Listing 2-18. A Person Class

```
// Person.java
package com.jdojo.binding;

import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;

public class Person {
        private ObjectProperty<Address> addr =
               new SimpleObjectProperty(new Address());

        public ObjectProperty<Address> addrProperty() {
                return addr;
        }
}
```

Suppose you create an ObjectProperty of the Person class as follows:

```
ObjectProperty<Person> p = new SimpleObjectProperty(new Person());
```

Using the Bindings.selectString() method, you can create a StringBinding for the zip property of the addr property of the Person object as shown here:

```
// Bind p.addr.zip
StringBinding zipBinding = Bindings.selectString(p, "addr", "zip");
```

The preceding statement gets a binding for the StringProperty zip, which is a nested property of the addr property of the object p. A property in the selectXXX() method may have multiple levels of nesting. You can have a selectXXX() call like

```
StringBinding xyzBinding = Bindings.selectString(x, "a", "b", "c", "d");
```

■ **Note** JavaFX API documentation states that `Bindings.selectString()` returns an empty `String` if any of its property arguments is inaccessible. However, the runtime returns `null`.

Listing 2-19 shows the use of the `selectString()` method. The program prints the values of the `zip` property twice: once for its default value and once for its changed value. At the end, it tries to bind a nonexistent property `p.addr.state`. Binding to a nonexistent property leads to an exception.

Listing 2-19. Using the `selectXXX()` Method of the `Bindings` Class

```
// BindNestedProperty.java
// Listing part of the example sources download for the book
```

```
36106
35217
null
Aug. 21, 2021 10:41:56 AM com.sun.javafx.binding.SelectBinding$SelectBindingHelper
getObservableValue
WARNING: Exception while evaluating select-binding [addr, state]
java.lang.NoSuchMethodException: com.jdojo.binding.BindNestedProperty$Address.getState()
     at java.base/java.lang.Class.getMethod(Class.java:2195)
     ...
     at  JavaFXBook/
     com.jdojo.binding.BindNestedProperty.main(BindNestedProperty.java:57)
```

Combining the Fluent API and the *Bindings* Class

While using the high-level binding API, you can use the fluent and `Bindings` class APIs in the same binding expression. The following snippet of code shows this approach:

```
DoubleProperty radius = new SimpleDoubleProperty(7.0);
DoubleProperty area = new SimpleDoubleProperty(0);

// Combine the Fluent API and Bindings class API
area.bind(Bindings.multiply(Math.PI, radius.multiply(radius)));
```

Using the Low-Level Binding API

The high-level binding API is not sufficient in all cases. For example, it does not provide a method to compute the square root of an `Observable` number. If the high-level binding API becomes too cumbersome to use or it does not provide what you need, you can use the low-level binding API. It gives you power and flexibility at the cost of a few extra lines of code. The low-level API allows you to use the full potential of the Java programming language to define bindings.

Using the low-level binding API involves the following three steps:

1. Create a class that extends one of the binding classes. For example, if you want to create a `DoubleBinding`, you need to extend the `DoubleBinding` class.
2. Call the `bind()` method of the superclass to bind all dependencies. Note that all binding classes have a `bind()` method implementation. You need to call this method passing all dependencies as arguments. Its argument type is a `varargs` of `Observable` type.
3. Override the `computeValue()` method of the superclass to write the logic for your binding. It calculates the current value of the binding. Its return type is the same as the type of the binding, for example, it is `double` for a `DoubleBinding`, `String` for a `StringBinding`, and so forth.

Additionally, you can override some methods of the binding classes to provide more functionality to your binding. You can override the `dispose()` method to perform additional actions when a binding is disposed. The `getDependencies()` method may be overridden to return the list of dependencies for the binding. Overriding the `onInvalidating()` method is needed if you want to perform additional actions when the binding becomes invalid.

Consider the problem of computing the area of a circle. The following snippet of code uses the low-level API to do this:

```
final DoubleProperty radius = new SimpleDoubleProperty(7.0);
DoubleProperty area = new SimpleDoubleProperty(0);

DoubleBinding areaBinding = new DoubleBinding() {
    {
        this.bind(radius);
    }

    @Override
    protected double computeValue() {
        double r = radius.get();
        double area = Math.PI * r * r;
        return area;
    }
};

area.bind(areaBinding); // Bind the area property to the areaBinding
```

The preceding snippet of code creates an anonymous class, which extends the `DoubleBinding` class. It calls the `bind()` method, passing the reference of the `radius` property. An anonymous class does not have a constructor, so you have to use an instance initializer to call the `bind()` method. The `computeValue()` method computes and returns the area of the circle. The `radius` property has been declared `final`, because it is being used inside the anonymous class.

The program in Listing 2-20 shows how to use the low-level binding API. It overrides the `computeValue()` method for the area binding. For the description binding, it overrides the `dispose()`, `getDependencies()`, and `onInvalidating()` methods as well.

Listing 2-20. Using the Low-Level Binding API to Compute the Area of a Circle

```
// LowLevelBinding.java
// Listing part of the example sources download for the book
```

```
Radius = 7.00, Area = 153.94
Description is invalid.
Radius = 14.00, Area = 615.75
```

Using Bindings to Center a Circle

Let's look at an example of a JavaFX GUI application that uses bindings. You will create a screen with a circle, which will be centered on the screen, even after the screen is resized. The circumference of the circle will touch the closer sides of the screen. If the width and height of the screen are the same, the circumference of the circle will touch all four sides of the screen.

Attempting to develop the screen, with a centered circle, without bindings is a tedious task. The `Circle` class in the `javafx.scene.shape` package represents a circle. It has three properties—`centerX`, `centerY`, and `radius`—of the `DoubleProperty` type. The `centerX` and `centerY` properties define the (x, y) coordinates of the center of the circle. The `radius` property defines the radius of the circle. By default, a circle is filled with black color.

You create a circle with `centerX`, `centerY`, and `radius` set to the default value of 0.0 as follows:

```
Circle c = new Circle();
```

Next, add the circle to a group and create a scene with the group as its root node as shown here:

```
Group root = new Group(c);
Scene scene = new Scene(root, 150, 150);
```

The following bindings will position and size the circle according to the size of the scene:

```
c.centerXProperty().bind(scene.widthProperty().divide(2));
c.centerYProperty().bind(scene.heightProperty().divide(2));
c.radiusProperty().bind(Bindings.min(scene.widthProperty(),
     scene.heightProperty()).divide(2));
```

The first two bindings bind the `centerX` and `centerY` of the circle to the middle of the width and height of the scene, respectively. The third binding binds the `radius` of the circle to the half (see `divide(2)`) of the minimum of the width and the height of the scene. That's it! The binding API does the magic of keeping the circle centered when the application is run.

Listing 2-21 has the complete program. Figure 2-5 shows the screen when the program is initially run. Figure 2-6 shows the screen when the screen is stretched horizontally. Try stretching the screen vertically and you will notice that the circumference of the circle touches only the left and right sides of the screen.

Listing 2-21. Using the Binding API to Keep a Circle Centered in a Scene

```
// CenteredCircle.java
// Listing part of the example sources download for the book
```

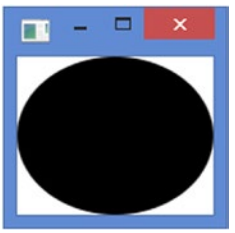

Figure 2-5. *The screen when the* `CenteredCircle` *program is initially run*

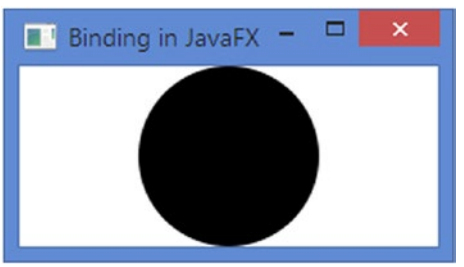

Figure 2-6. *The screen when the screen for the* `CenteredCircle` *program is stretched horizontally*

Summary

A Java class may contain two types of members: fields and methods. Fields represent the state of its objects, and they are declared private. Public methods, known as accessors, or getters and setters, are used to read and modify private fields. A Java class having public accessors for all or part of its private fields is known as a Java bean, and the accessors define the properties of the bean. Properties of a Java bean allow users to customize its state, behavior, or both.

JavaFX supports properties, events, and binding through properties and binding APIs. Properties support in JavaFX is a huge leap forward from the JavaBeans properties. All properties in JavaFX are observable. They can be observed for invalidation and value changes. You can have read/write or read-only properties. All read/write properties support binding. In JavaFX, a property can represent a value or a collection of values.

A property generates an invalidation event when the status of its value changes from valid to invalid for the first time. Properties in JavaFX use lazy evaluation. When an invalid property becomes invalid again, an invalidation event is not generated. An invalid property becomes valid when it is recomputed.

In JavaFX, a binding is an expression that evaluates to a value. It consists of one or more observable values known as its dependencies. A binding observes its dependencies for changes and recomputes its value automatically. JavaFX uses lazy evaluation for all bindings. When a binding is initially defined or when its dependencies change, its value is marked as invalid. The value of an invalid binding is computed when it is requested next time. All property classes in JavaFX have built-in support for binding.

A binding has a direction, which is the direction in which changes are propagated. JavaFX supports two types of binding for properties: unidirectional binding and bidirectional binding. A unidirectional binding works only in one direction; changes in dependencies are propagated to the bound property, not vice versa. A bidirectional binding works in both directions; changes in dependencies are reflected in the property and vice versa.

The binding API in JavaFX is divided into two categories: high-level binding API and low-level binding API. The high-level binding API lets you define binding using the JavaFX class library. For most use cases, you can use the high-level binding API. Sometimes, the existing API is not sufficient to define a binding. In those cases, the low-level binding API is used. In the low-level binding API, you derive a binding class from an existing binding class and write your own logic to define the binding.

The next chapter will introduce you to observable collections in JavaFX.