Application Program Development

Segment : Properties and Bidning

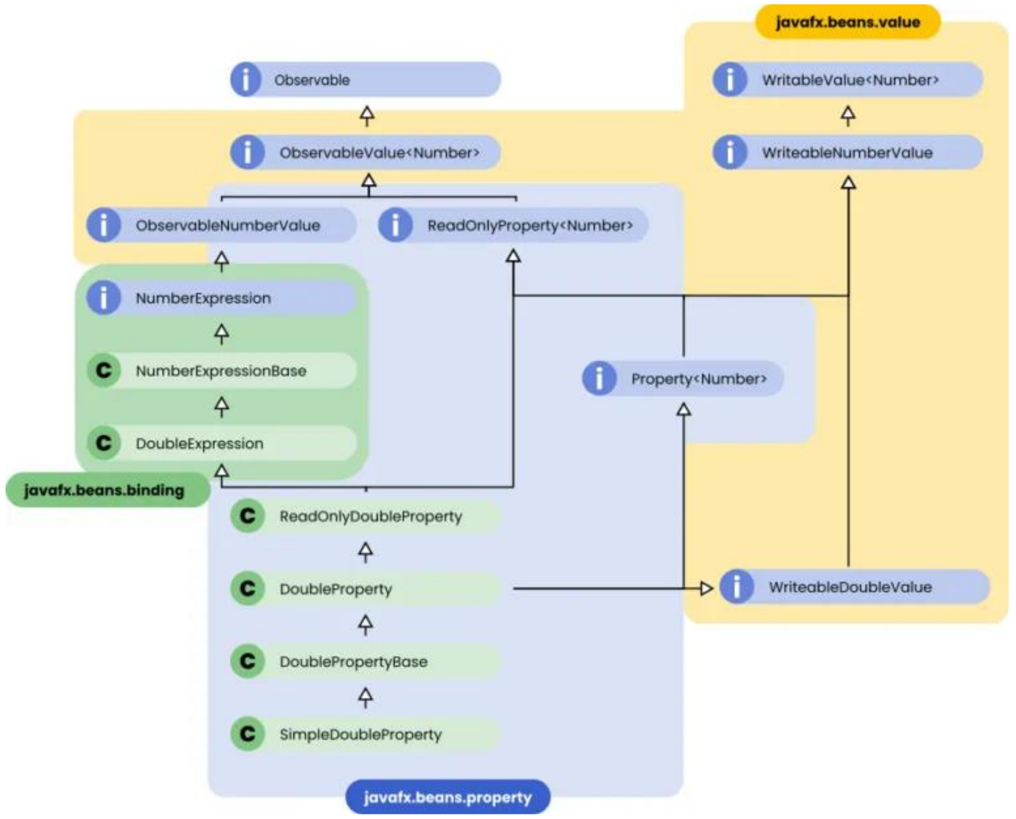
Mahboob Ali

Outcomes

- Understanding What are
  - JavaFx properties.
  - JavaFx binding.
  - Need of properties and their bindings

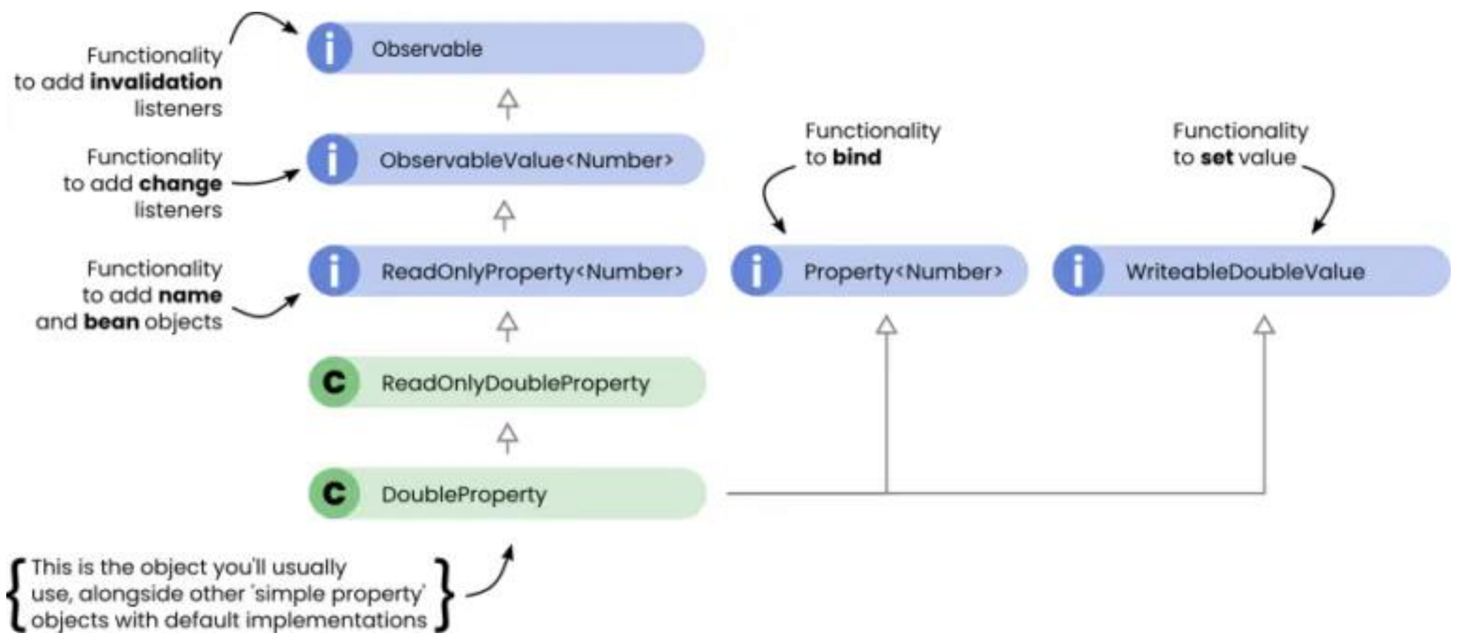
Introduction

- Properties are basically wrapper objects for JavaFX-based object attributes such as String or Integer.
- Properties allow developers to add listener code to respond when the wrapped value of an object has changed or is flagged as invalid. Also, property objects can be bound to one another.
- Binding behavior allows properties to update or synchronize their values based on a changed value from another property.
- Among other things, properties let you wire your scene so that the view updates automatically whenever you modify the data that sits behind it.
- Properties are observable objects that may be writeable, or read-only.
- There are 30 types of Property object in JavaFX, including the StringProperty, SimpleListProperty and ReadOnlyObjectProperty.
- Each property wraps an existing Java object, adding functionality for listening and binding.
- In addition to the properties functionality, JavaFX provides functionality for binding values through Binding and Expression objects.
- Binding is a mechanism for enforcing relationships between objects, in which one or more observable objects are used to update the value of another object.
- Bindings can act in one, or both directions and can be created either directly from properties (the Fluent API) or using the Bindings utility class (Bindings API).
- Custom Bindings objects can also be created manually if you need extra customization or performance. This is called the Low-Level API.



Types of Properties

- Properties and binding are a group of interfaces and classes designed to make your life as a developer significantly easier.
- That being said, with 61 properties, and 249 methods in the Bindings class, it *can* get overwhelming and difficult to manage.
- In *JavaFX's Properties API* there are two types to be concerned about:
  - Read/writable
  - Read-only
- In short, JavaFX's properties are wrapper objects that hold actual values while providing change support, invalidation support, and binding capabilities.
- Properties are wrapper objects that have the ability to make values accessible as read/writable or readonly.
- All wrapper property classes are located in the javafx.beans.property.* package namespace.


Functionality to add **invalidation** listeners
Observable
Functionality to add **change** listeners
ObservableValue<Number>
Functionality to add **name** and **bean** objects
ReadOnlyProperty<Number>
Functionality to **bind**
Property<Number>
Functionality to **set** value
WriteableDoubleValue
ReadOnlyDoubleProperty
DoubleProperty
This is the object you'll usually use, alongside other 'simple property' objects with default implementations

What is a property?

- There's nothing magic under under the bonnet, though.
- Most of the JavaFX Property objects extend two key interfaces:
    - ReadOnlyProperty<T> and WriteableValue<T>.
- Some of them don't, though. JavaFX has 10 read-only properties, which extend ReadOnlyProperty<T>, but don't extend WriteableValue<T>.

Read/ Writeable Properties

- Read/writable properties are, as the name suggests, property values that can be both read and modified.
- As an example, let's look at JavaFX string properties.
- To create a string property that is capable of both readable and writable access to the wrapped value, you will use the javafx.beans. property.

```
StringProperty password = new SimpleStringProperty("password1");

password.set("1234");

System.out.println("Modified StringProperty " + password.get() ); // 1234
```

- In the case of reading the value back, you would invoke the get() method (or getValue()), which returns the actual wrapped value(String) to the caller.
- To modify the value, you simply call the set() method (or setValue()) by passing in a string.

Read-Only Properties

- To create a property to be read-only, you need to take two steps.
  - First, you instantiate a read-only wrapper class.
  - Second, invoke the method getReadOnlyProperty() to return a true read-only property object.

```
ReadOnlyStringWrapper userName = new ReadOnlyStringWrapper("jamestkirk");
ReadOnlyStringProperty readOnlyUserName = userName.getReadOnlyProperty();
```

- According to the Javadoc API, the *ReadOnlyStringWrapper* class “creates two properties that are synchronized.
- One property is read-only and can be passed to external users.
- The other property is readable and writable and should be used internally only.
- ” Knowing that the other property can be read and written could allow a malicious developer to cast the object to type StringProperty, which then could be modified at will.
- From a security perspective, you should be aware of the proper steps to create a true read-only property.

Creating a property

- JavaFX comes with ten in-built classes that make creating a property significantly easier. They implement all of the required functionality, from listening to binding.

SimpleBooleanProperty	**SimpleLongProperty**
SimpleDoubleProperty	SimpleMapProeprty
SimpleFloatProeprty	SimpleObjectProperty
SimlpeIntegerProperty	SimpleSetProeprty
SimpleListProperty	SimpleStringProperty

- You can define any of the simple property objects either with or without an initial value. If defined without the default value, they'll default to the default value of the object the property wraps – 0, false, "" or an empty collection.

```
SimpleIntegerProperty()
SimpleIntegerProperty(int initialValue)
```

- They can also be created with a name, and an Object that JavaFX refers to as the property's "bean". This doesn't encapsulate the property in any way, but creates a symbolic link to an object that represents the property's "owner".

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

Creating a property

- Neither the name nor bean attributes changes the behavior of the property, but it can act as a useful look-up.
- That's useful if you're attaching the same listener to multiple properties – especially properties generated programmatically.
- Then, once a change occurs, you can use the bean and name attributes to check which property just changed.
- All JavaFX properties have methods to facilitate the following functions:
    1. Listening for changes to the property's value
    2. Wring properties together (binding) so that they update automatically
    3. Getting and setting (if writable) the property value

JavaFX Java Bean: Example

```java
import javafx.beans.property.*;
public class User {
        private final static String USERNAME_PROP_NAME = "userName";
        private final ReadOnlyStringWrapper userName;
        private final static String PASSWORD_PROP_NAME = "password";
        private final StringProperty password;
        public User() {
                userName = new ReadOnlyStringWrapper(this,
                USERNAME_PROP_NAME, System.getProperty("user.name"));
                password = new SimpleStringProperty(this, PASSWORD_PROP_NAME, "");
        }
        public final String getUserName() {
                return userName.get();
        }
        public ReadOnlyStringProperty userNameProperty() {
                return userName.getReadOnlyProperty();
        }
        public final String getPassword() {
                return password.get();
        }
        public final void setPassword(String password) {
                this.password.set(password);
        }
        public StringProperty passwordProperty() {
                return password;
        }
}
```

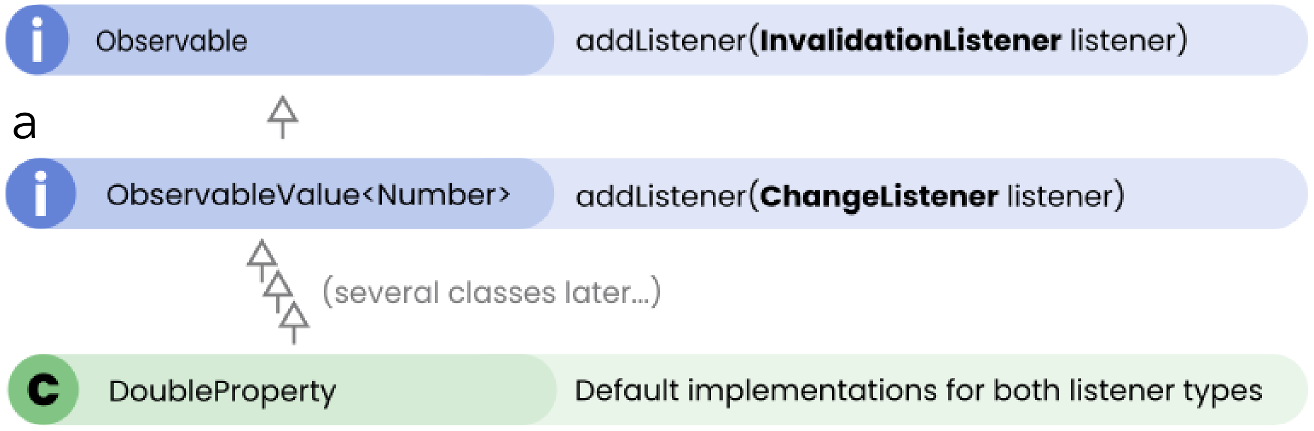
- You may notice some obvious differences in the way I instantiated the `ReadOnlyStringWrapper` and `SimpleStringProperty` classes.
- Similar to the JavaBean property change support, JavaFX properties have constructors that will allow you to specify the bean itself, its property name, and its value.

How to observe the change in a property?

- JavaFX Property objects are a mish-mash of different implemented interfaces.
- ***Property change support*** is the ability to add handler code that will respond when a property changes.
- JavaFX property objects contain an `addListener()` method. This method will accept two types of functional interfaces:
  - ChangeListener
  - InvalidationListener
- Functional interfaces are single abstract methods that are expressed using the Java lambda syntax.
- All JavaFX properties are descendants of the `ObservableValue` and `Observable interfaces` (method overloading), which provide the `addListener()` methods for `ChangeListener` and `InvalidationListener`, respectively.
- Finally, remember that it is important to clean up listeners by removing them.
- The purpose of cleaning up listeners is to reclaim memory.
- To remove them, you will invoke the `removeListener()` method by passing into it a referenced (named) listener as opposed to an anonymous inner class or anonymous lambda expression

Invalidation Listeners:

- Every property in JavaFX extends the Observable interface, meaning they all provide functionality to register listeners that fire when the property invalidates.
- If you're not familiar with 'invalidates', it's a way of marking a property as potentially changed without forcing it to recalculate the property's value.
- For properties with complex or expensive calculations, that can a useful tool, but I don't find they're used as much as change listeners.


Observable
addListener(**InvalidationListener** listener)
ObservableValue<Number>
addListener(**ChangeListener** listener)
(several classes later...)
DoubleProperty
Default implementations for both listener types

Invalidation Listeners

- As the property's value changes, the invalidated () method will be invoked.
- I've provided the implementations for both the anonymous inner class and lambda expression-style syntax for you to compare.

```
// Adding a invalidation listener (anonymous inner class)
xProperty.addListener(new InvalidationListener() {
@Override
public void invalidated(Observable o) {
       // code goes here
}
});
// Adding a invalidation listener (lambda expression)
xProperty.addListener((Observable o) -> {
       // code goes here
});
```

Change Listeners

- On top of that, JavaFX properties extend ObservableValue<T>, meaning you can register listeners that only fire when an object has actually changed.
- We use these a lot more regularly than invalidation listeners.
- Change listeners allow us to hear a change, and provide executable code ahead of time, which will execute based on the old and new values of the Property.
- Adding a ChangeListener Instance to Respond When the Property Value Changes

```
SimpleIntegerProperty xProperty = new SimpleIntegerProperty(0);
       // Adding a change listener (anonymous inner class)
       xProperty.addListener(new ChangeListener<Number>(){
       @Override
       public void changed(ObservableValue<? extends Number> ov, Number oldVal,
                                                        Number newVal) {
              // code goes here
       }
       });
       // Adding a change listener (lambda expression)
       xProperty.addListener((ObservableValue<? extends Number> ov, Number oldVal,
                                                  Number newVal) -> {
              // code goes here
       });
```

Difference between Change & Invalidation Listeners

- Using a `ChangeListener` you will get the Observable (ObservableValue), the old value, and the new value,
- Using the `InvalidationListener` only gets the Observable object (property).
- The `InvalidationListener` provides a way to mark values as invalid but does not recompute the value until it is needed.
- This is often used in UI layouts or custom controls, where you can avoid unnecessary computations when nodes don't need to be redrawn/repositioned during a layout request or draw cycle.
- When using the `ChangeListener`, you normally want eager evaluation such as the validation of properties on a form-type application.
- That doesn't mean you can't use `InvalidationListeners` for validation of properties; it just depends on your performance requirements and exactly when you need the new value to be recomputed (evaluated).
- When you access the observable value, it causes the `InvalidationListener` to be eager.

Binding

- Binding has the idea of at least two values (properties) being synchronized.
- This means that when a dependent variable changes, the other variable changes.
- JavaFX provides many binding options that enable the developer to synchronize between properties in domain objects and GUI controls.
- Properties can be bound natively themselves, as well as by creating Expression and Binding objects.
- Expressions and Bindings are observable objects that also depend on the value of at least one other observable object (but potentially more).
- That enables you to create expression chains with multiple calculations: a super-simple way to string together string or number conversions.

How to Bind Properties

- Behind the scenes, binding is a specific use-case of change-listening.

- All of JavaFX’s binding APIs have boilerplate code that listens for a change in (at least) one property and uses any change to update the value of that binding.

- Where change listeners let us provide executable code ahead of time, binding gives us the convenience of stringing two properties together without having to worry about the implementation of updating a particular value.

- The simplest and most heavily used are the methods that come attached to Property objects themselves:
  - bind()
  - bindBidirectional().

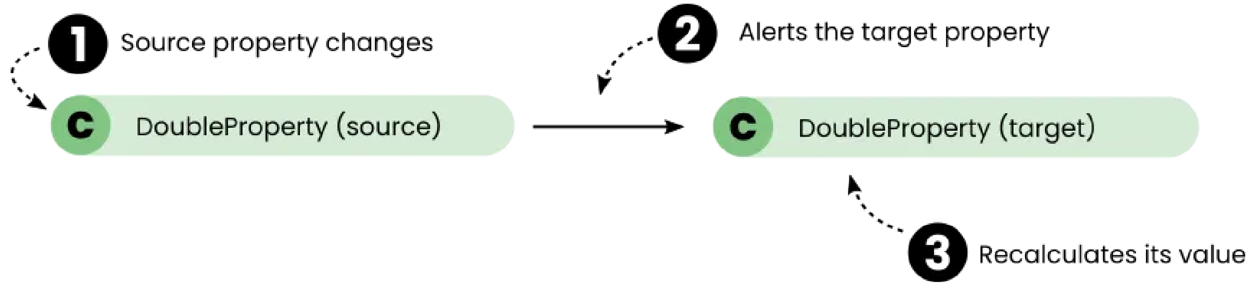
- These represent the simplest options for one-way and two-way bindings.

One-directional binding

- When you invoke the `bind()` method on a target property, you pass it a second property as an argument – the binding source

```
    StringProperty sourceProperty = new SimpleStringProperty("First Value");
    StringProperty targetProperty = new SimpleStringProperty("Second Value");
    targetProperty.bind(sourceProperty);
```

- Under the bonnet, the target stores a reference to the new source property and listens for changes.
- **When the source value changes, it automatically updates the target (itself) when a change is detected.**
- In this case, targetProperty will track the value of sourceProperty. A few notes on extra bits for this method:
    - If the property is currently bound, the current binding is deleted and the new one replaces it
    - If a null argument is provided, the method throws a NullPointerExeption
    - The method immediately copies the value of the property it's listening to, so the current value of the target property is lost.

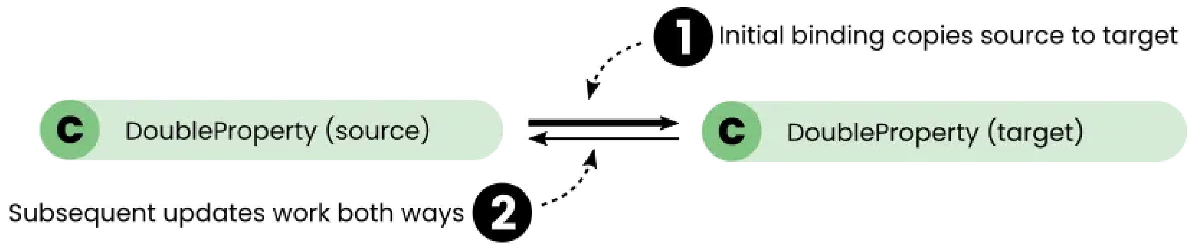

1 Source property changes
C DoubleProperty (source)
2 Alerts the target property
C DoubleProperty (target)
3 Recalculates its value

Two-directional binding

- It is, of course, possible to wire the properties together in both directions, linking their values together so that their values are always the same.
- To accomplish this, we invoke bindBidirectional(), passing the source property as an argument again.

```
StringProperty sourceProperty = new SimpleStringProperty("First Value");
StringProperty targetProperty = new SimpleStringProperty("Second Value");
targetProperty.bindBidirectional(sourceProperty);
```

- If the values of the properties are different, *then the order of the method call is important in determining the start value for the binding.*
- The method applied to `targetProperty` immediately updates the value of `targetProperty` before `sourceProperty` is bound reciprocally.
- That means that **after a bidirectional binding, both properties will have the value of the property passed as an argument (the source property).**
- In addition to the basic binding, which is limited to a carbon-copy approach, JavaFX supports more complex binding: *creating multiple or complex manipulations of one property to update another.*


1
Initial binding copies source to target
C
DoubleProperty (source)
C
DoubleProperty (target)
Subsequent updates work both ways
2

Intermediate binding techniques

- There are three ways to manipulate any property and used that manipulated value for binding:

- The Fluent API – methods like myProperty.bind(otherProperty).multiply(2)
- The Bindings API – methods like Bindings.add(myProperty, otherProperty)
- The Low-Level API – creating custom Bindings objects like DoubleBinding

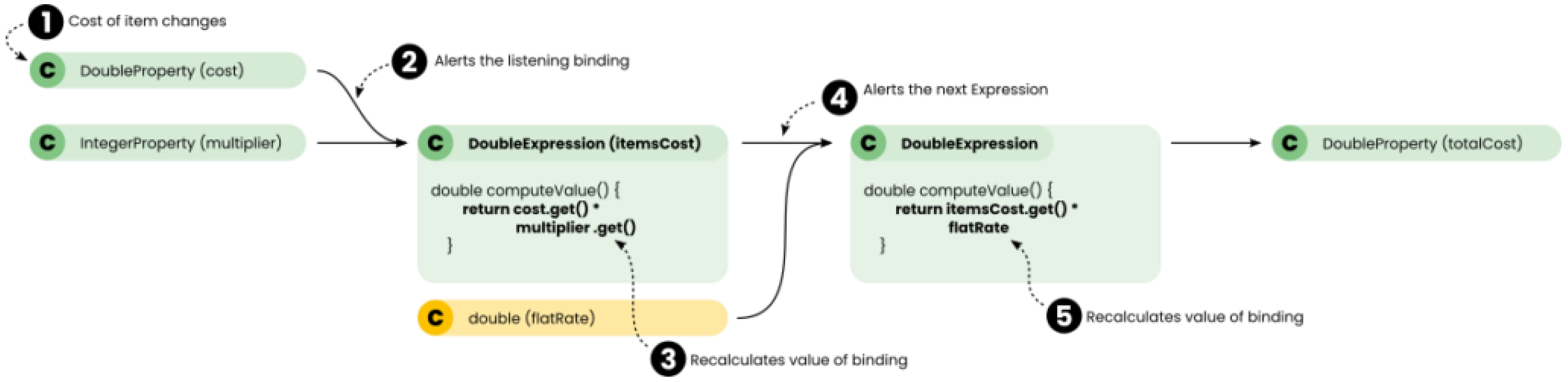
- Two of these provide cookie-cutter methods to beind properties in pre-defined implementations.
- I've always found these cover the majority of use cases for property binding, because they give you a huge amount of flexibilty.

- The lower-level API – creating Binding objects – can be higher performance but can get a lot more complicated.

Fluent API

- The Fluent API relies on the creation of "expression" objects, which are similar to properties (they're observable values) with extra convenience methods to support extra manipulation.

- Properties can also be bound to expressions, meaning the output of any manipulations can be used to update a property, just as above. That functionality – of being observable *and* depending on an object for a value – creates the possibility of chaining.


1 Cost of item changes
C DoubleProperty (cost)
C IntegerProperty (multiplier)
2 Alerts the listening binding
C DoubleExpression (itemsCost)
double computeValue() {
return cost.get() *
multiplier .get()
}
C double (flatRate)
3 Recalculates value of binding
4 Alerts the next Expression
C DoubleExpression
double computeValue() {
return itemsCost.get() *
flatRate
}
5 Recalculates value of binding
C DoubleProperty (totalCost)

Fluent API

- In the case of Strings, we can use this to create chains of Strings, which are concatenated together. As soon as the `sourceProperty` is updated, the `targetProperty` will be updated automatically, via the expression.

```
StringProperty sourceProperty = new SimpleStringProperty(
                                             "It doesn't matter how slowly you go");
StringExpression expression = sourceProperty.concat(" as long as you don't stop");
StringProperty targetProperty = new SimpleStringProperty("");
targetProperty.bind(expression);
System.out.println(targetProperty.get());
//Output: It doesn't matter how slowly you go as long as you don't stop
```

- You can do all of this in-line, making complex code relatively concise. In this case, we'll create the `StringExpression` while we call the bind method.

```
targetProperty.bind(sourceProperty.concat(" as long as you don't stop"));
System.out.println(targetProperty.get());

//Output: It doesn't matter how slowly you go as long as you don't stop
```

Fluent API

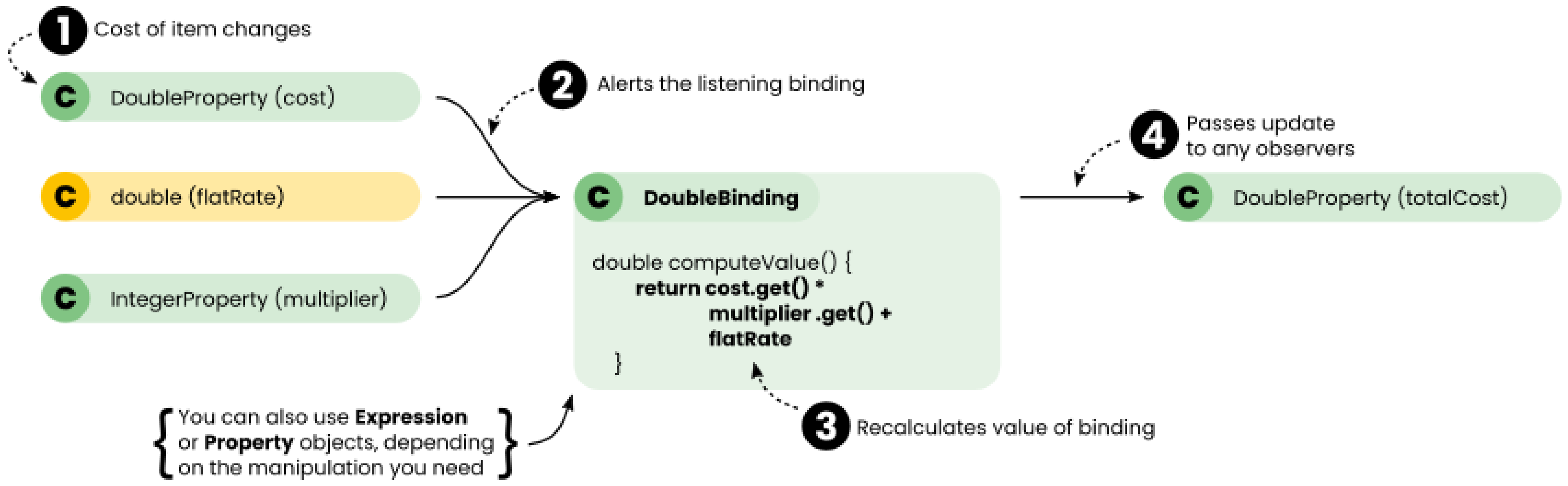
- This can be a little confusing, but don't forget targetProperty is actually bound to a `StringExpression` created by the `concat()` method.
- It's the anonymous expression that gets bound to the `sourceProperty`.
- This comes with amazing benefits. The API is rich, and the style produces readable code that covers the majority of manipulations you'll need.
- We can also use the Fluent API with numbers.
- With numbers, we can chain manipulations to create simple, readable code that represents the formula we're trying to replicate. To convert from degrees to radians, you multiply by Pi and divide by 180. The code is highly readable.

```
DoubleProperty degrees = new SimpleDoubleProperty(180);
DoubleProperty radians = new SimpleDoubleProperty();
radians.bind(degrees
               .multiply(Math.PI)
              .divide(180)
     );
```

- In terms of performance, every expression is a link in a chain that needs to be updated on each change of the initial property. In the above example, where we convert from degrees to radians, we're creating two observable values just to update our radians property.

The Bindings API

- The Bindings class in JavaFX is a utility class containing 249 methods for property binding.
- It allows you to create bindings between observable objects of various types.
- You can combine properties with values, such a Strings and numbers, depending on the binding.



1 Cost of item changes
C DoubleProperty (cost)
C double (flatRate)
C IntegerProperty (multiplier)
2 Alerts the listening binding
C DoubleBinding
4 Passes update to any observers
C DoubleProperty (totalCost)
3 Recalculates value of binding
You can also use Expression or Property objects, depending on the manipulation you need

```
double computeValue() {
    return cost.get() *
        multiplier .get() +
        flatRate
  }
```

The Bindings API

- There are 10 general binding strategies, which you can divide in two main areas
  - Operations on values
  - Operations on collections.
  - Other (which doesn't fit any where).

Values	Collections	Other bindings
Mathematics (+, - , /, x)	Binding two collections (lists, sets, maps)	Multiple-object bindings
Selecting maximum or minimum	Binding values to objects at certain position in a collection	Boolean operations (and, not, or, when)
Comparisons (=, !=, <, >, <=, >=)	Binding to collection size	Selecting values
String formatting	Whether a collection is empty	

The Bindings API – Operations on values

- The Bindings API provides support for four simple mathematical operations:
    - addition
    - subtraction
    - multiplication
    - division.

- It provides separate methods for use of these with float, double, integer and long values, as well as between two `ObservableNumberValue` objects (for example, a `DoubleProperty`).

```
DoubleBinding    add(double op1, ObservableNumberValue op2)
NumberBinding    add(float op1, ObservableNumberValue op2)
NumberBinding    add(int op1, ObservableNumberValue op2)
NumberBinding    add(long op1, ObservableNumberValue op2)
DoubleBinding    add(ObservableNumberValue op1, double op2)
NumberBinding    add(ObservableNumberValue op1, float op2)
NumberBinding    add(ObservableNumberValue op1, int op2)
NumberBinding    add(ObservableNumberValue op1, long op2)
NumberBinding    add(ObservableNumberValue op1, ObservableNumberValue op2)
```

The Bindings API – Operations on values

- The same methods are available for each of the numerical options.
- The API swaps out the first and second arguments for convenience.
- For addition and multiplication that doesn't make a difference (order doesn't matter), but with subtraction, the order determines which argument is subtracted from the other.

```
DoubleBinding    subtract(double op1, ObservableNumberValue op2)
DoubleBinding    subtract(ObservableNumberValue op1, double op2)
NumberBinding    subtract(ObservableNumberValue op1, ObservableNumberValue op2)
```

- In each case, the second argument is subtracted from the first

```
DoubleBinding    multiply(double op1, ObservableNumberValue op2)
DoubleBinding    multiply(ObservableNumberValue op1, double op2)
NumberBinding    multiply(ObservableNumberValue op1, ObservableNumberValue op2)
```

- Finally, with division, order becomes important again. The value of the binding is calculated as the first argument divided by the second argument.

```
DoubleBinding    divide(double op1, ObservableNumberValue op2)
DoubleBinding    divide(ObservableNumberValue op1, double op2)
NumberBinding    divide(ObservableNumberValue op1, ObservableNumberValue op2)
```

The Bindings API – Operations on Collections

- The Bindings API gives you four different ways to bind to a collection:
  - carbon-copy
  - index-binding
  - size-binding
  - emptiness-binding.

- Only the first one copies the contents of your collection into a target collection.

- The other three extract values – single variables – based on one aspect of the collection's state.

- To bind two collections together, you can invoke either `Bindings.bindContent()`, or `Bindings.bindContentBidirectional()`.

- In the first case, you'll be tracking an observable collection – an ObservableList, ObservableSet or ObservableMap – and creating a carbon-copy in a non-observable collection of the same type.

```
bindContent(List<E> list1, ObservableList<? extends E> list2)
bindContent(Map<K,V> map1, ObservableMap<? extends K,? extends V> map2)
bindContent(Set<E> set1, ObservableSet<? extends E> set2)
```

Advanced Binding Tech – Low-Level Binding

- The most customizable way to create a binding in JavaFX is to manually create a Binding object yourself.
- The benefits of this are that you get to define exactly the calculations you want, and you don't have to create chains of Expression objects, which can reduce performance.
- Of course, you can also do complex calculations and bind multiple objects with the Bindings API, but as we saw above, that's not quite as efficient.
- The benefit of the Low-Level API is that any calculations you need to perform when calculating the value is defined inside your custom Binding class.
- If you want the technical terms for why this is likely to be better performing, it's that class functions are more likely to be 'in-lined' by the compiler.
- That means your code will probably perform faster if you need the value of a binding to be computer repeatedly and quickly.

What is Low-Level Binding?

- The Low-Level API, a collection of 10 abstract Binding classes designed to implement all of the awkward bits of binding (adding and removing listeners, for example).
- That frees you to concentrate on specifying how the value of the binding should be calculated.
- Each class takes observable values (like properties) and converts them into an output.
- Just like the Fluent and Bindings APIs, the Low-Level API provides support for boolean, string, numbers, collections and objects.

Creating a Low-Level Binding

- Creating a Low-Level binding can be as simple as defining an abstract inner class (a class you define alongside other code).
- Because the abstract Bindings classes only have one abstract method, you only need to override `computeValue()` when you define the method.
- As you define the binding, the official advice is to use an initialization block that binds up the source properties during binding creation.
- The overall effect is exactly the same – the compiler copies the code from initialization blocks into every constructor anyway.
- The constructor approach is more appropriate if you're creating a concrete class you're going to use more than once.

```
//Inside your binding object at the top
  {
              super.bind(cost, itemCount);
 }
```

Creating a Low-Level Binding

- Then all that's left is to define the `computeValue()` method. In this case, it's pretty simple, but you can make the calculation as complicated as you want.

```
      DoubleProperty cost = new SimpleDoubleProperty(25);
      IntegerProperty itemCount = new SimpleIntegerProperty(15);
      DoubleBinding totalCost = new DoubleBinding() {

        {
                 super.bind(cost, itemCount);
            }

      @Override
      protected double computeValue() {
               return itemCount.get() * cost.get();
            }
      };
```

- From this point out, the value of the `totalCost` binding will always reflect the product of the cost and itemCount properties.

- If you want to be able to pass the `totalCost` object around and retrieve the dependencies later, you can add extra functionality to override the default `getDependencies()` method.