Behavior Summary

There are three techniques for implementing behavior for controls and menu items. They all involve implementing a function, often as a function literal. It can be used to set the value of a handler property of a control or menu item or registered as an invalidation listener for a state property of a control.

- event handlers — are functions assigned to an event handler property such as the `onAction` property of many controls and menu components. When the user performs a certain action on the control or menu component the method is invoked.
- invalidation listeners — are functions registered with a state property of a control by being passed as an argument to the property's `addListener()` method. The method is invoked whenever the value of the property changes.
- bindings — are a simplified mechanism for registering invalidation listeners that can be used in simple cases.

Properties

JavaFX JavaDoc for a class often contains a "Property Summary" section. This section lists properties for objects in the class. A *property* is an extension of the Java Bean convention. It has a Bean value and an additional method for accessing the property object:

- `get`*`Name`*`()` — returns the value for the property named by *name*.
- `set`*`Name`*`()` (optional) — sets the value for the property named by *name*.
- *`name`*`Property()` — returns the property object named by *name*.

The property object is more than just a value. It has its own methods. For example, if a control captures its state in a property then the property object will have an `addListener()` method. This is used for registering a function to be executed when the property value changes.

For implementing behavior, properties are used in two important ways:

- *Event handler properties* have functions as values. The functions, often function literals, determine how controls respond to user actions.
- *State properties* capture information entered or selected by users.

Function Literals

The simplest function literal is a *lambda expression*, which has the following form:

```
(/* typed parameter list goes here */) -> {
  // Code to implement the function.
}
```

Lambda expressions can be assigned as values for event handler properties. The values of these properties implement the EventHandler<*T*> interface for some event type *T*. Many controls and menu items have an `onAction` property with a value of type EventHandler<ActionEvent>. Then the lambda expression has the following more specific form:

```
(ActionEvent ev) -> {
  // Code to implement the function.
}
```

For example, the following code sets the `onAction` handler for a check box with a function that displays the state of the check box in its label.

```
theCheckBox.setOnAction((ActionEvent ev) -> {
  String stateText = (theCheckBox.isSelected() ? "" : "un") + "checked";
  theCheckBox.setText(stateText);
});
```

Lambda expressions can be used for invalidation listeners. Invalidation listeners implement the InvalidationListener interface. This interface has a single method with an Observable parameter. Then the lambda expression has the following more specific form:

```
(Observable obs) -> {
  // Code to respond to a change in the value of obs.
}
```

For example, the following code registers an InvalidationListener with the `value` state property of a slider.

```
ObservableValue<Number> value = theSlider.valueProperty();
value.addListener((Observable obs) -> {
  sliderLabel.setText(value.getValue().toString());
});
```

Whenever the slider value changes its new value is displayed in a label. This label could be placed to give the user feedback on the slider value.

Bindings

All JavaFX properties have a `bind()` method. This method has a parameter of interface type ObservableValue<*value-type*>, where *value-type* is the same type as the type of the property value. Conveniently, all JavaFX properties also implement this interface. Consequently any property can be bound to another property of the same type.

If you execute a statement with the following form

```
property.bind(observableValue);
```

where the property and the observable value have matching types, the effect is that the property value tracks the observable value. This has two implications:

- The property getter returns the same value as the observable value.
- The property passes invalidations from the observable value to its own invalidation listeners.

Bindings use a simple caching mechanism to do this as efficiently as possible.