



# CS-202 Introduction to Object Oriented Programming

*California State University, Los Angeles  
Computer Science Department*

Lecture 14  
**Event-Driven Programming**  
**Slides by Keenan Knaur**

# Introduction

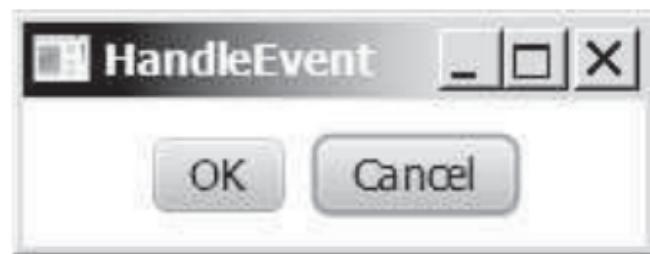
# Introduction

- ➊ This week we will see how to add actions to GUI controls in order to make them actually do something
  - i.e. click a button and do some action.
- ➋ There are many types of actions:
  - mouse clicks
  - key presses
  - mouse movement
  - etc.

# Introduction

- See Code: `HandleEvent.java`

- This program creates a window with two buttons, and clicking on them displays messages to the console.



(a)



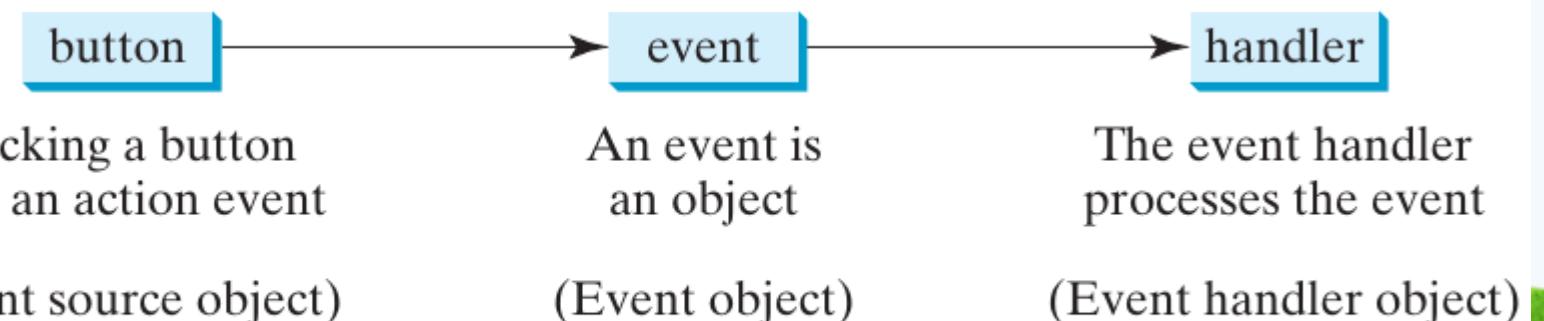
```
C:\book>java HandleEvent
OK button clicked
Cancel button clicked
OK button clicked
Cancel button clicked
```

(b)

**FIGURE 15.2** (a) The program displays two buttons. (b) A message is displayed in the console when a button is clicked.

# Introduction

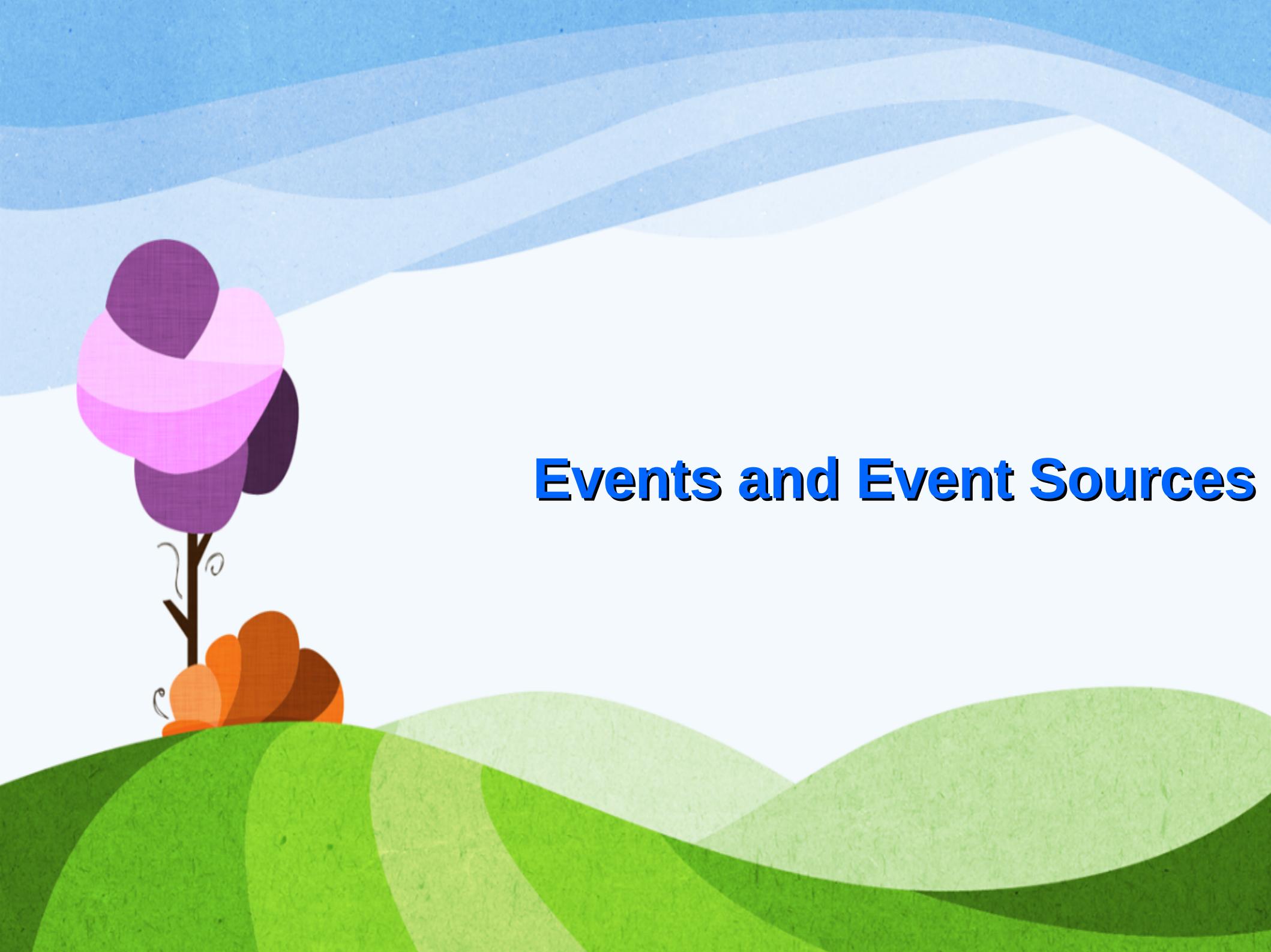
- For button clicks to work, you need to write the "back-end" code which processes the button click action.
- We say that the button is an ***event source object*** (the object from which an event or action originates)
- You need to create an object which "handles" the action event on a button
  - this is called an ***event handler***



**FIGURE 15.3** An event handler processes the event fired from the source object.

# Introduction

- Not all objects can be handlers for an action event
- To be a handler:
  - The object must be an instance of the **EventHandler<T extends Event>** interface.
    - ◆ NOTE: **<T extends Event>** means that **T** is a generic subtype of **Event**.
  - The **EventHandler** object must be "registered" with the event source object using the method **source.setOnAction(handler)**.
  - - ◆ You also need to override the **handle(ActionEvent)** method to respond to the event.

The background features a stylized landscape with green rolling hills at the bottom, a white central area, and blue wavy patterns at the top. A small tree with a brown trunk and purple/pink leaves sits on a green hill on the left side.

# **Events and Event Sources**

# Events and Event Sources

- ➊ **event**: object created from an event source, a signal to the program that something has happened.
- ➋ **firing an event**: creating an event and delegating the handler to handle the event.
- ➌ a user interacts with a GUI program and the events of the GUI are what drive its execution (this is called **event-driven programming**)
- ➍ objects which create and fire events are called **event source objects**, **source objects**, or **source components**

# Events and Event Sources

- All events are instances of an event class.
- The root of all Java event classes is `java.util.EventObject`
- The root class of JavaFX event classes is `javafx.event.Event`.

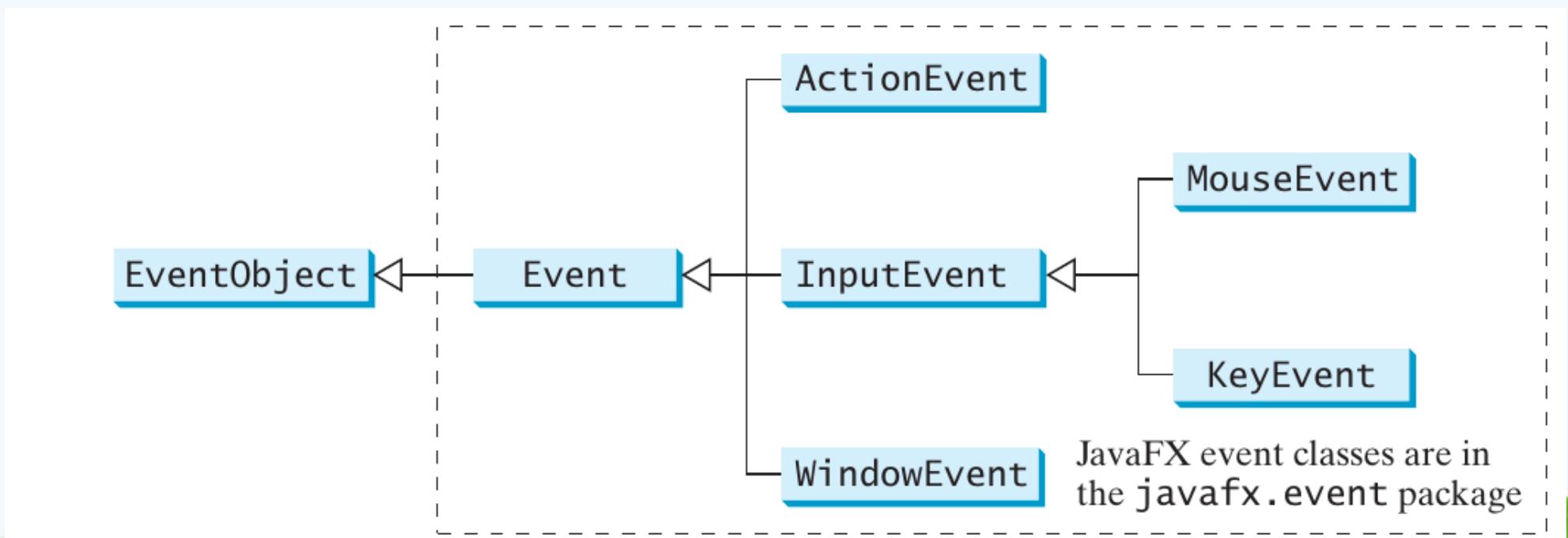


FIGURE 15.4 An event in JavaFX is an object of the `javafx.event.Event` class.

# Events and Event Sources

- ➊ **event objects** contain whatever properties are important to the event
- ➋ the `getSource()` method of the `EventObject` class can be used to identify the source objects.
- ➋ Any subclass of a component that can fire an event will also fire the same type of event.

**TABLE 15.1** User Action, Source Object, Event Type, Handler Interface, and Handler

User Action	Source Object	Event Type Fired	Event Registration Method
Click a button	<b>Button</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Press Enter in a text field	<b>TextField</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Check or uncheck	<b>RadioButton</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Check or uncheck	<b>CheckBox</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Select a new item	<b>ComboBox</b>	<b>ActionEvent</b>	<code>setOnAction(EventHandler&lt;ActionEvent&gt;)</code>
Mouse pressed	<b>Node, Scene</b>	<b>MouseEvent</b>	<code>setOnMousePressed(EventHandler&lt;MouseEvent&gt;)</code>
Mouse released			<code>setOnMouseReleased(EventHandler&lt;MouseEvent&gt;)</code>
Mouse clicked			<code>setOnMouseClicked(EventHandler&lt;MouseEvent&gt;)</code>
Mouse entered			<code>setOnMouseEntered(EventHandler&lt;MouseEvent&gt;)</code>
Mouse exited			<code>setOnMouseExited(EventHandler&lt;MouseEvent&gt;)</code>
Mouse moved			<code>setOnMouseMoved(EventHandler&lt;MouseEvent&gt;)</code>
Mouse dragged			<code>setOnMouseDragged(EventHandler&lt;MouseEvent&gt;)</code>
Key pressed	<b>Node, Scene</b>	<b>KeyEvent</b>	<code>setOnKeyPressed(EventHandler&lt;KeyEvent&gt;)</code>
Key released			<code>setOnKeyReleased(EventHandler&lt;KeyEvent&gt;)</code>
Key typed			<code>setOnKeyTyped(EventHandler&lt;KeyEvent&gt;)</code>



# **Registering Handlers & Handling Events**

# Registering Handlers and Handling Events

- A handler is an object that has to be registered with an event source object. Must also be an instance of an appropriate event-handling interface.

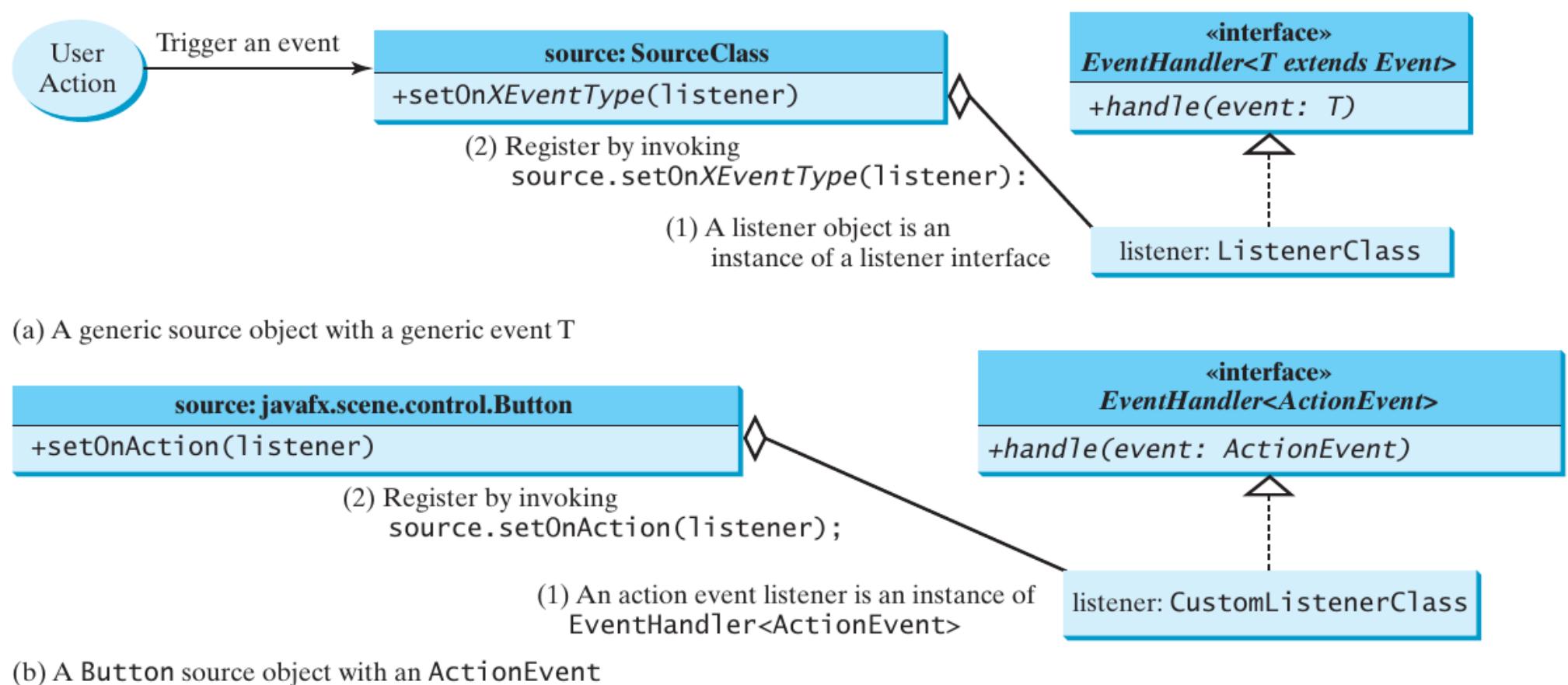


FIGURE 15.5 A listener must be an instance of a listener interface and must be registered with a source object.

# Registering Handlers and Handling Events

- See Code:

- `ControlCircleWithoutEventHandler.java`
  - `ControlCircle.java`

# Inner Classes



# Inner Classes

- also called *nested classes*, are classes defined within the scope of another class.
- useful for defining handler classes.

```
public class Test {  
    ...  
  
    public class A {  
        ...  
    }  
}
```

(a)

```
public class Test {  
    ...  
  
    // Inner class  
    public class A {  
        ...  
    }  
}
```

(b)

```
// OuterClass.java: inner class demo  
public class OuterClass {  
    private int data;  
  
    /** A method in the outer class */  
    public void m() {  
        // Do something  
    }  
  
    // An inner class  
    class InnerClass {  
        /** A method in the inner class */  
        public void mi() {  
            // Directly reference data and method  
            // defined in its outer class  
            data++;  
            m();  
        }  
    }  
}
```

(c)

FIGURE 15.7 Inner classes combine dependent classes into the primary class.

# Inner Classes

- ➊ Can be used just like regular classes.
- ➋ Normally defined to only be used by the class they are defined in.
  - If you want to make an inner class accessible to other classes, then just define it in its own class.
- ➌ Compile into a class called  
**OuterClassName\$InnerclassName.class**
- ➍ Can reference data and methods defined in the outer class
  - you don't need to pass anything through constructors or method parameters between the outer and inner classes.

# Inner Classes

- ➊ can be defined with a visibility modifier (uses the same rules as previous)
- ➋ can be defined as static
  - static inner classes can be accessed using the outer class name
- ➌ To create an object of an inner class from another class use the following if the inner class is non static

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

- ➍ If the inner class is static use:

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

# **Anonymous Inner Classes**

# Anonymous Inner Classes

- An inner class without a name, combines defining an inner class AND creating an instance of the class all in one step.
- Used as a way to shorten inner-class handlers

```
public void start(Stage primaryStage) {  
    // Omitted  
  
    btEnlarge.setOnAction(  
        new EnlargeHandler());  
}  
  
class EnlargeHandler  
    implements EventHandler<ActionEvent> {  
    public void handle(ActionEvent e) {  
        circlePane.enlarge();  
    }  
}
```

(a) Inner class EnlargeListener

```
public void start(Stage primaryStage) {  
    // Omitted  
  
    btEnlarge.setOnAction(  
        new class EnlargeHandler  
            implements EventHandler<ActionEvent>() {  
                public void handle(ActionEvent e) {  
                    circlePane.enlarge();  
                }  
            }  
    );  
}
```

(b) Anonymous inner class

# Anonymous Inner Classes

- ⦿ Treated like an inner class with the following properties:
  - must always extend a superclass or implement an interface, but cannot have an explicit extends or implements clause.
  - must implement all the abstract methods in the superclass or interface.
  - always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is Object().
  - compiled into a class named `OuterClassName$n.class`.
    - ◆ if the outer class Test has two anonymous inner classes, they are compiled into `Test$1.class` and `Test$2.class`.
- ⦿ See Code: `AnonymousHandlerDemo.java`

# Lambda Expressions



# Lambda Expressions

- another way to simplify coding for event handling (as if programmers aren't lazy enough sheesh!)
- new feature added to Java 8
- are like an anonymous class with a very concise syntax.

```
btEnlarge.setOnAction(  
    new EventHandler<ActionEvent>() {  
        @Override  
        public void handle(ActionEvent e) {  
            // Code for processing event e  
        }  
    }  
);
```

(a) Anonymous inner class event handler

```
btEnlarge.setOnAction(e -> {  
    // Code for processing event e  
});
```

(b) Lambda expression event handler

# Lambda Expressions

- Basic Syntax:

```
(type1 param1, type2 param2, ...) -> expression
```

or

```
(type1 param1, type2 param2, ...) -> { statements; }
```

- the datatype for a parameter can be explicitly declared, or implicitly inferred by the compiler.
- the parenthesis can be omitted if there is only one parameter without an explicit data type

# Lambda Expressions

- The compiler treats a lambda expression like it is an object created from an anonymous inner class.
- In the previous example: the compiler understands that the object must be an instance of **EventHandler<ActionEvent>**.
- The **EventHandler** interface defines only one method, **handle()**, with a parameter of the **ActionEvent** type.
  - the compiler can infer that e should be a parameter of the **ActionEvent** type
  - any statements in the lambda expression are part of the body of the **handle()** method.



# Lambda Expressions

- ➊ If the interface contained more than one method it would not be able to compile the lambda expression.
  - the requirement is that the interface must contain exactly one abstract method.
  - this kind of interface is known as a ***functional interface*** or a ***Single Abstract Method (SAM)*** interface.
- ➋ See Code:
  - `LambdaHandlerDemo.java`
  - `LoanCalculator.java`

# Mouse Events

# Mouse Events

- fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or scene.
- **MouseEvent** objects capture the event and properties related to that event:
  - number of clicks
  - x, y location of the mouse
  - which mouse button was pressed

# Mouse Events

<code>javafx.scene.input.MouseEvent</code>	
<code>+getButton(): MouseButton</code>	Indicates which mouse button has been clicked.
<code>+getClickCount(): int</code>	Returns the number of mouse clicks associated with this event.
<code>+getX(): double</code>	Returns the <i>x</i> -coordinate of the mouse point in the event source node.
<code>+getY(): double</code>	Returns the <i>y</i> -coordinate of the mouse point in the event source node.
<code>+getSceneX(): double</code>	Returns the <i>x</i> -coordinate of the mouse point in the scene.
<code>+getSceneY(): double</code>	Returns the <i>y</i> -coordinate of the mouse point in the scene.
<code>+getScreenX(): double</code>	Returns the <i>x</i> -coordinate of the mouse point in the screen.
<code>+getScreenY(): double</code>	Returns the <i>y</i> -coordinate of the mouse point in the screen.
<code>+isAltDown(): boolean</code>	Returns true if the Alt key is pressed on this event.
<code>+isControlDown(): boolean</code>	Returns true if the Control key is pressed on this event.
<code>+isMetaDown(): boolean</code>	Returns true if the mouse Meta button is pressed on this event.
<code>+isShiftDown(): boolean</code>	Returns true if the Shift key is pressed on this event.

FIGURE 15.10 The `MouseEvent` class encapsulates information for mouse events.

- Also defines four constants **PRIMARY**, **SECONDARY**, **MIDDLE**, and **NONE** to indicate the left, right, middle, and none mouse buttons.
- See Code: `MouseEventDemo.java`

# **Key Events**

# Key Events

- A **KeyEvent** is fired whenever a key is pressed, released, or typed on a node or scene.

## `javafx.scene.input.KeyEvent`

```
+getCharacter(): String  
+getCode(): KeyCode  
+getText(): String  
+isAltDown(): boolean  
+isControlDown(): boolean  
+isMetaDown(): boolean  
+isShiftDown(): boolean
```

Returns the character associated with the key in this event.  
Returns the key code associated with the key in this event.  
Returns a string describing the key code.  
Returns true if the Alt key is pressed on this event.  
Returns true if the Control key is pressed on this event.  
Returns true if the mouse Meta button is pressed on this event.  
Returns true if the Shift key is pressed on this event.

**FIGURE 15.12** The **KeyEvent** class encapsulates information about key events.

# Key Events

- Every key event has an associated code that is returned by the `getCode()` method
  - these codes are constants (see next table)
- For the key-pressed and key-released events, `getCode()` returns the value as defined in the table, `getText()` returns a string that describes the key code, and `getCharacter()` returns an empty string.
- For the key-typed event, `getCode()` returns `UNDEFINED` and `getCharacter()` returns the Unicode character or a sequence of characters associated with the key-typed event.

# Key Events

**TABLE 15.2** KeyCode Constants

Constant	Description	Constant	Description
<b>HOME</b>	The Home key	<b>CONTROL</b>	The Control key
<b>END</b>	The End key	<b>SHIFT</b>	The Shift key
<b>PAGE_UP</b>	The Page Up key	<b>BACK_SPACE</b>	The Backspace key
<b>PAGE_DOWN</b>	The Page Down key	<b>CAPS</b>	The Caps Lock key
<b>UP</b>	The up-arrow key	<b>NUM_LOCK</b>	The Num Lock key
<b>DOWN</b>	The down-arrow key	<b>ENTER</b>	The Enter key
<b>LEFT</b>	The left-arrow key	<b>UNDEFINED</b>	The <b>keyCode</b> unknown
<b>RIGHT</b>	The right-arrow key	<b>F1</b> to <b>F12</b>	The function keys from F1 to F12
<b>ESCAPE</b>	The Esc key	<b>0</b> to <b>9</b>	The number keys from 0 to 9
<b>TAB</b>	The Tab key	<b>A</b> to <b>Z</b>	The letter keys from A to Z

See Code: **KeyEventDemo.java**

# **Listeners for Observable Objects**



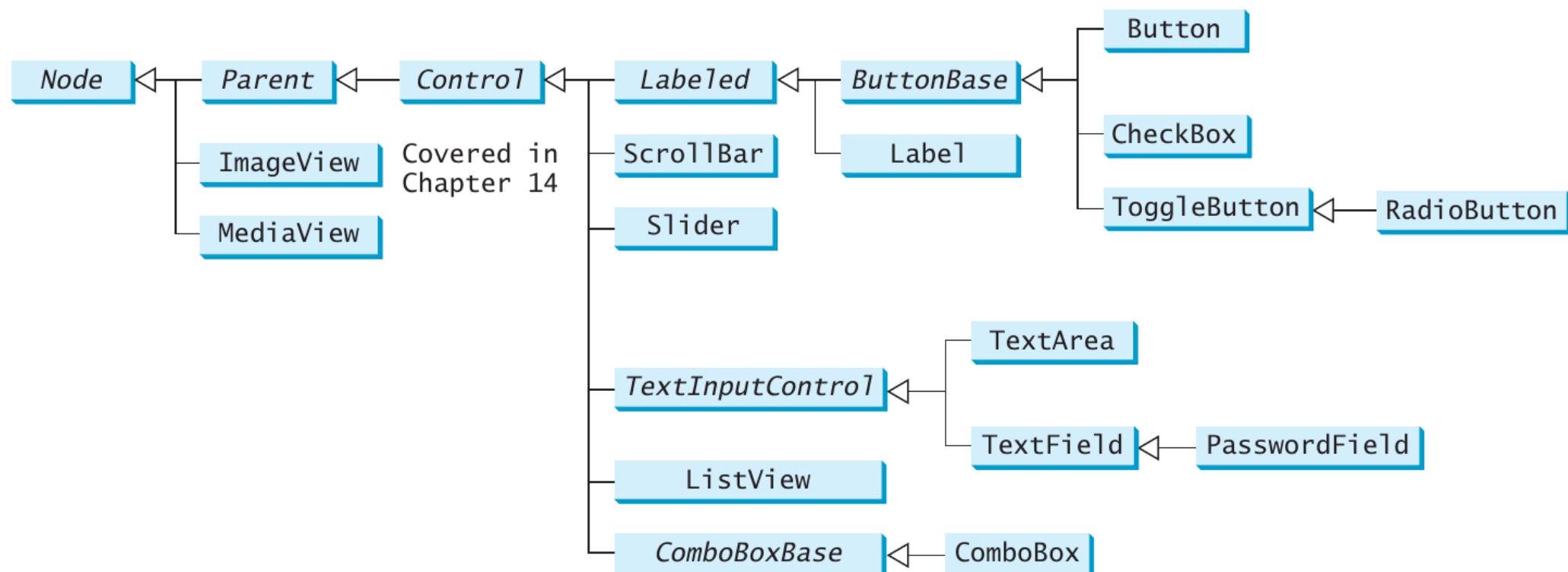
# Listeners for Observable Objects

- Listeners can be added to observable objects.
- A observable object is an instance of **Observable** and contains the **addListener(InvalidationListener listener)** method for adding a listener.
- The listener class must implement the **InvalidationListener** interface and override the **invalidated(Observable o)** method for handling the value change.
- If the value is changed in the **Observable** object, the listener is notified by invoking the **invalidated(Observable o)** method.
- All binding properties are instances of **Observable**.
- See Code:
  - **ObservablePropertyDemo.java**
  - **DisplayResizeableClock.java**

The background features a stylized landscape with green rolling hills at the bottom, a white central area, and blue wavy patterns at the top. A small, whimsical tree with a brown trunk and branches stands on the left, its leaves composed of overlapping circles in shades of purple, pink, and dark purple.

# JavaFX UI Controls

# Common JavaFX UI Controls



**FIGURE 16.1** These UI controls are frequently used to create user interfaces.

# Labeled and Label

- a **label** is used to display short text, a node, or both
- usually used to label a textfield.
- The **Labeled** class defines many common properties.

*javafx.scene.control.Labeled*

```
-alignment: ObjectProperty<Pos>
-contentDisplay:
    ObjectProperty<ContentDisplay>
-graphic: ObjectProperty<Node>
-graphicTextGap: DoubleProperty
-textFill: ObjectProperty<Paint>
-text: StringProperty
-underline: BooleanProperty
-wrapText: BooleanProperty
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies the alignment of the text and node in the labeled.  
Specifies the position of the node relative to the text using the constants **TOP**, **BOTTOM**, **LEFT**, and **RIGHT** defined in **ContentDisplay**.  
A graphic for the labeled.  
The gap between the graphic and the text.  
The paint used to fill the text.  
A text for the labeled.  
Whether text should be underlined.  
Whether text should be wrapped if the text exceeds the width.

FIGURE 16.2 **Labeled** defines common properties for **Label**, **Button**, **CheckBox**, and **RadioButton**.

# Labeled and Label

*javafx.scene.control.Labeled*



**javafx.scene.control.Label**

+Label()  
+Label(text: String)  
+Label(text: String, graphic: Node)

Creates an empty label.

Creates a label with the specified text.

Creates a label with the specified text and graphic.

**FIGURE 16.3** **Label** is created to display a text or a node, or both.

➊ See Code: **LabelWithGraphic.java**

# Button

<i>javafx.animation.Animation</i>	
<p>-autoReverse: BooleanProperty -cycleCount: IntegerProperty -rate: DoubleProperty -status: ReadOnlyObjectProperty   &lt;Animation.Status&gt;</p> <p>+pause(): void +play(): void +stop(): void</p>	<p>The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.</p> <p>Defines whether the animation reverses direction on alternating cycles. Defines the number of cycles in this animation. Defines the speed and direction for this animation. Read-only property to indicate the status of the animation.</p> <p>Pauses the animation. Plays the animation from the current position. Stops the animation and resets the animation.</p>

**FIGURE 15.15** The abstract **Animation** class is the root class for JavaFX animations.

See Code: **ButtonDemo.java**

# CheckBox

- inherits from **ButtonBase** and **Labeled**
- when a check box is clicked (checked or unchecked) it fires an **ActionEvent**.
- use **isSelected()** to see if the checkbox is selected.
- See Code: **CheckBoxDemo.java**

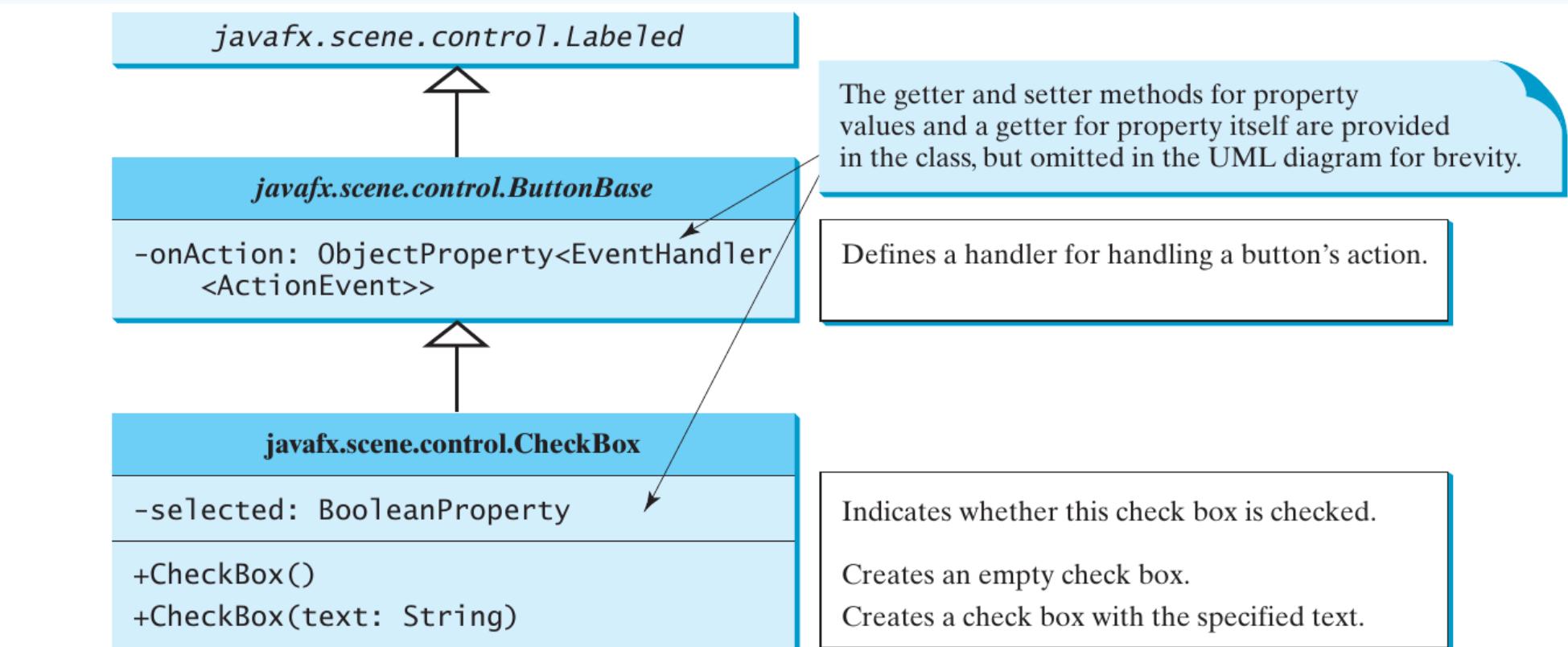


FIGURE 16.7 **CheckBox** contains the properties inherited from **ButtonBase** and **Labeled**.

# RadioButton

- Subclass of ToggleButton

- RadioButton is displayed as a circle
- ToggleButton is displayed similar to a button.

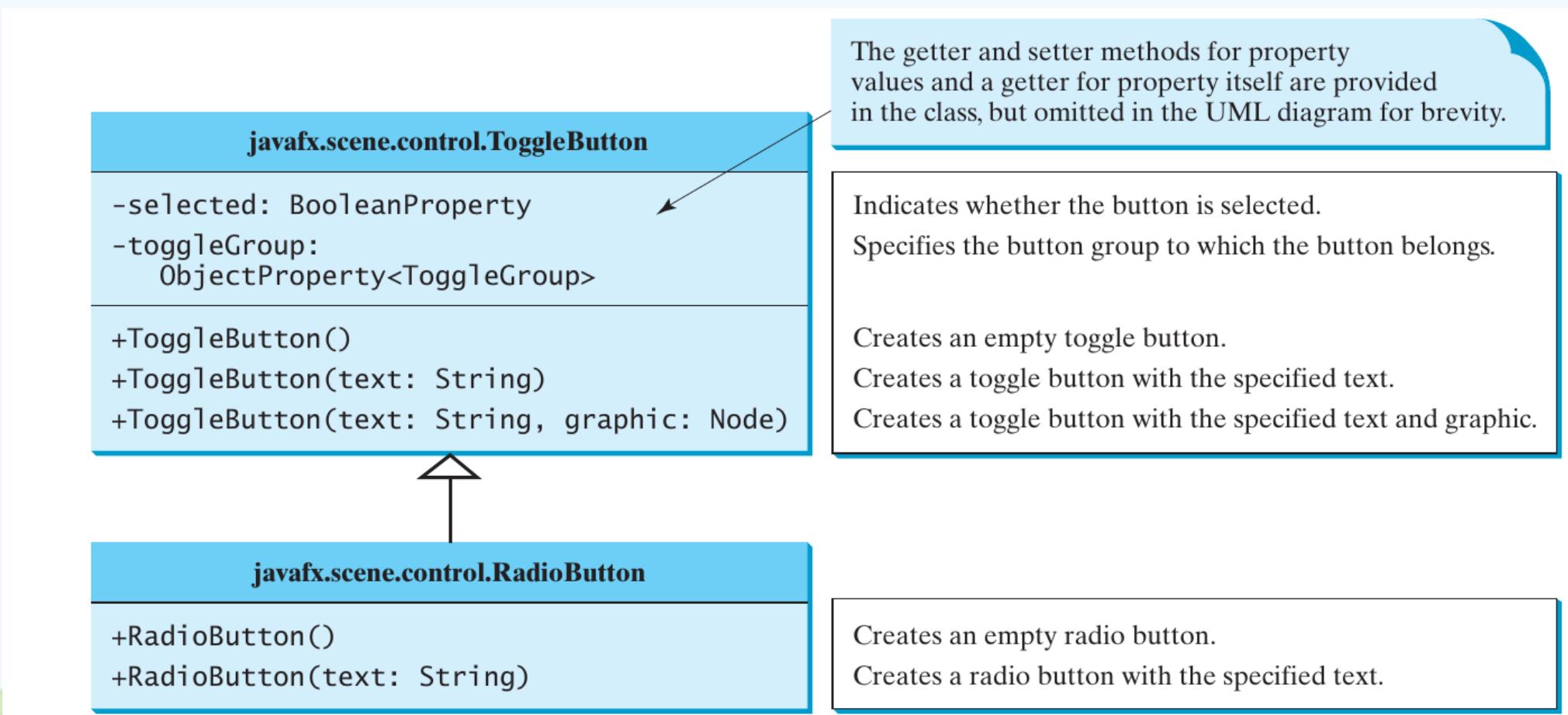


FIGURE 16.9 `ToggleButton` and `RadioButton` are specialized buttons for making selections.

# RadioButton

```
RadioButton rbUS = new RadioButton("US");
rbUS.setGraphic(new ImageView("image/usIcon.gif"));
rbUS.setTextFill(Color.GREEN);
rbUS.setContentDisplay(ContentDisplay.LEFT);
rbUS.setStyle("-fx-border-color: black");
rbUS.setSelected(true);
rbUS.setPadding(new Insets(5, 5, 5, 5));
```



To group radio buttons, you need to create an instance of **ToggleGroup** and set a radio button's **toggleGroup** property to join the group, as follows:

```
ToggleGroup group = new ToggleGroup();
rbRed.setToggleGroup(group);
rbGreen.setToggleGroup(group);
rbBlue.setToggleGroup(group);
```

● See Code: **RadioButtonDemo.java**

# TextField

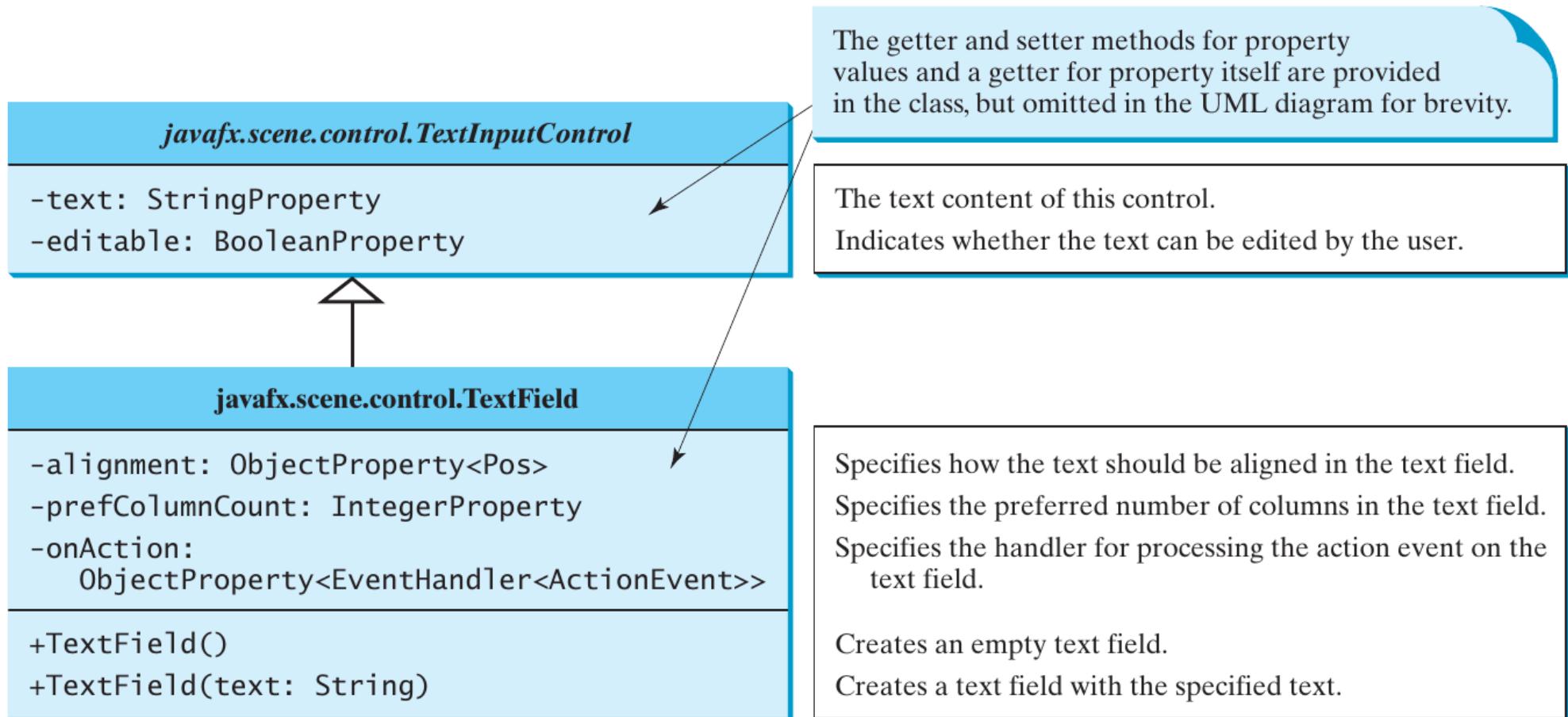


FIGURE 16.11 `TextField` enables the user to enter or display a string.

➊ See `TextFieldDemo.java`

# TextArea

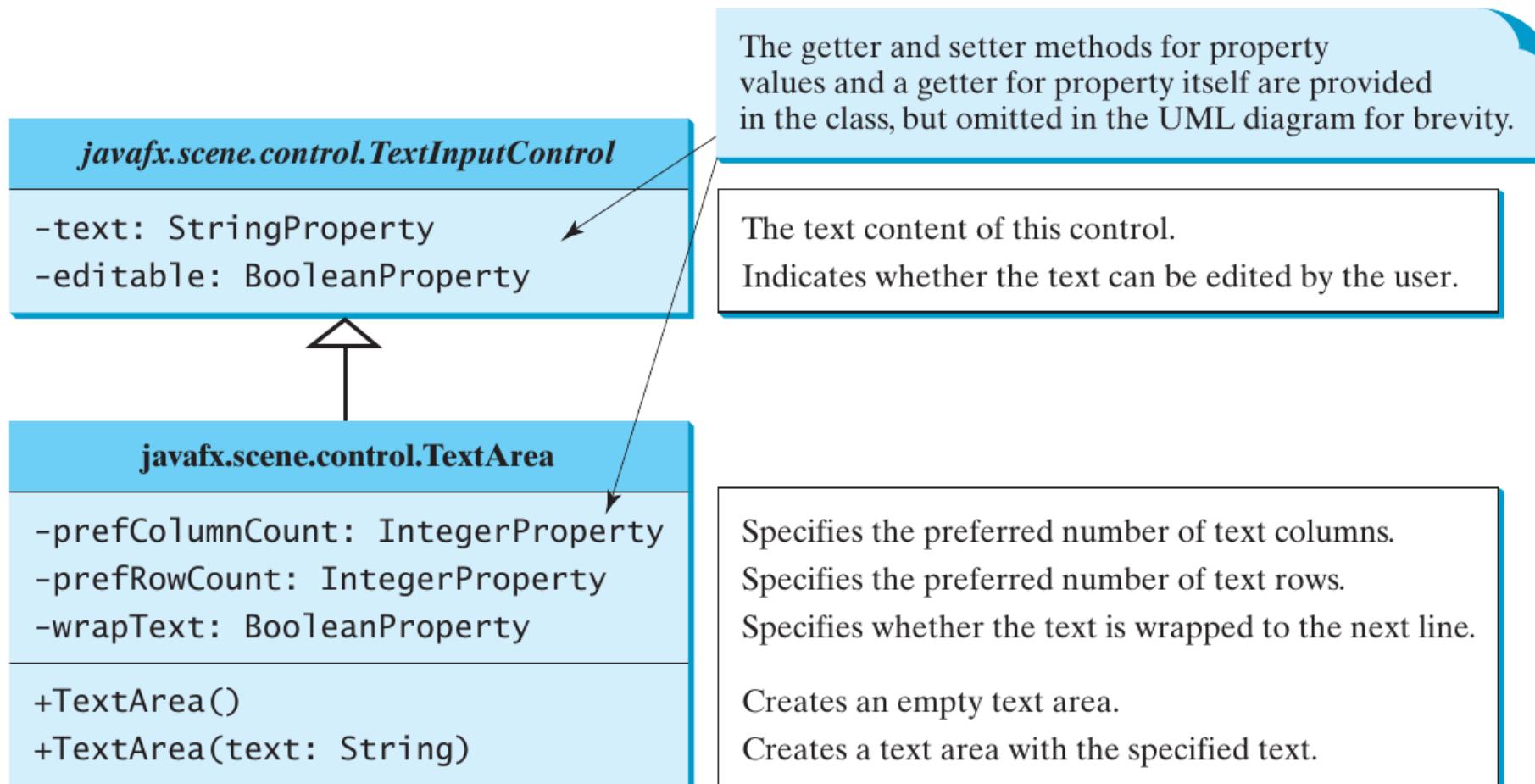


FIGURE 16.13 `TextArea` enables the user to enter or display multiple lines of characters.

See Code: `TextAreaDemo.java`

# ComboBox

- Presents a drop-down list of items the user can choose from.
- Can fire an **ActionEvent** whenever an item is selected.
- See Code: **ComboBoxDemo.java**

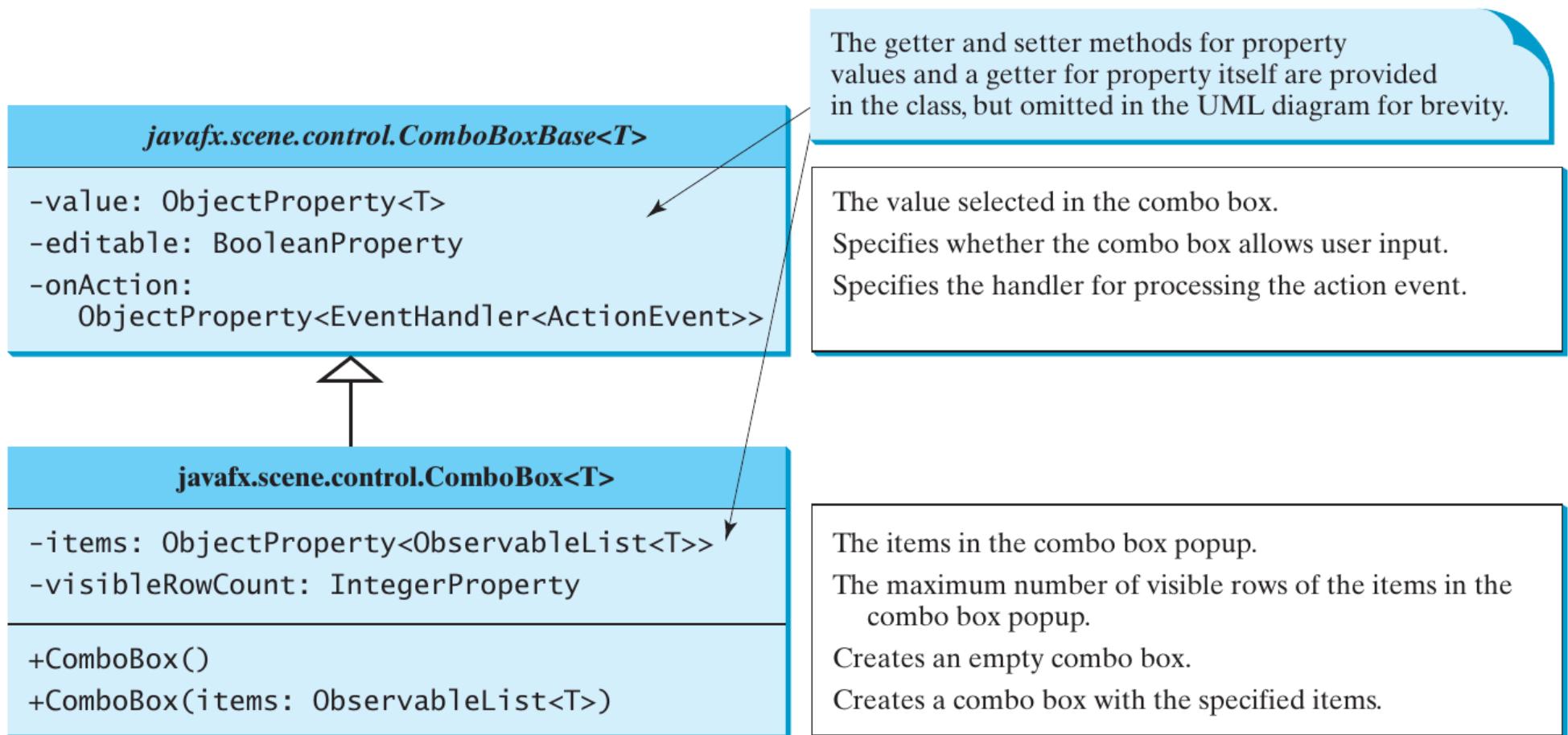


FIGURE 16.16 `ComboBox` enables the user to select an item from a list of items.

# ListView

- Same functionality as a ComboBox, but allows the user to choose one or more values.
- See `ListViewDemo.java`

`javafx.scene.control.ListView<T>`

-items: ObjectProperty<ObservableList<T>>  
-orientation: BooleanProperty  
  
-selectionModel:  
ObjectProperty<MultipleSelectionModel<T>>

+ListView()  
+ListView(items: ObservableList<T>)

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The items in the list view.  
Indicates whether the items are displayed horizontally or vertically in the list view.  
Specifies how items are selected. The SelectionModel is also used to obtain the selected items.

Creates an empty list view.  
Creates a list view with the specified items.

FIGURE 16.18 `ListView` enables the user to select one or multiple items from a list of items.

# ScrollBar

## See Code: ScrollBarDemo.java

javafx.scene.control.ScrollBar	The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
-blockIncrement: DoubleProperty	The amount to adjust the scroll bar if the track of the bar is clicked (default: 10).
-max: DoubleProperty	The maximum value represented by this scroll bar (default: 100).
-min: DoubleProperty	The minimum value represented by this scroll bar (default: 0).
-unitIncrement: DoubleProperty	The amount to adjust the scroll bar when the <code>increment()</code> and <code>decrement()</code> methods are called (default: 1).
-value: DoubleProperty	Current value of the scroll bar (default: 0).
-visibleAmount: DoubleProperty	The width of the scroll bar (default: 15).
-orientation: ObjectProperty<Orientation>	Specifies the orientation of the scroll bar (default: HORIZONTAL).
+ScrollBar()	Creates a default horizontal scroll bar.
+increment()	Increments the value of the scroll bar by <code>unitIncrement</code> .
+decrement()	Decrements the value of the scroll bar by <code>unitIncrement</code> .

FIGURE 16.22 **ScrollBar** enables the user to select from a range of values.

# Slider

- Similar to a **ScrollBar** but has more properties and can appear in many forms.
- See Code:
  - SliderDemo.java**
  - BounceBallSlider.java**

javafx.scene.control.Slider	
-blockIncrement: DoubleProperty	The amount to adjust the slider if the track of the bar is clicked (default: 10).
-max: DoubleProperty	The maximum value represented by this slider (default: 100).
-min: DoubleProperty	The minimum value represented by this slider (default: 0).
-value: DoubleProperty	Current value of the slider (default: 0).
-orientation: ObjectProperty<Orientation>	Specifies the orientation of the slider (default: HORIZONTAL).
-majorTickUnit: DoubleProperty	The unit distance between major tick marks.
-minorTickCount: IntegerProperty	The number of minor ticks to place between two major ticks.
-showTickLabels: BooleanProperty	Specifies whether the labels for tick marks are shown.
-showTickMarks: BooleanProperty	Specifies whether the tick marks are shown.
+Slider()	Creates a default horizontal slider.
+Slider(min: double, max: double, value: double)	Creates a slider with the specified min, max, and value.

FIGURE 16.25 **Slider** enables the user to select from a range of values.