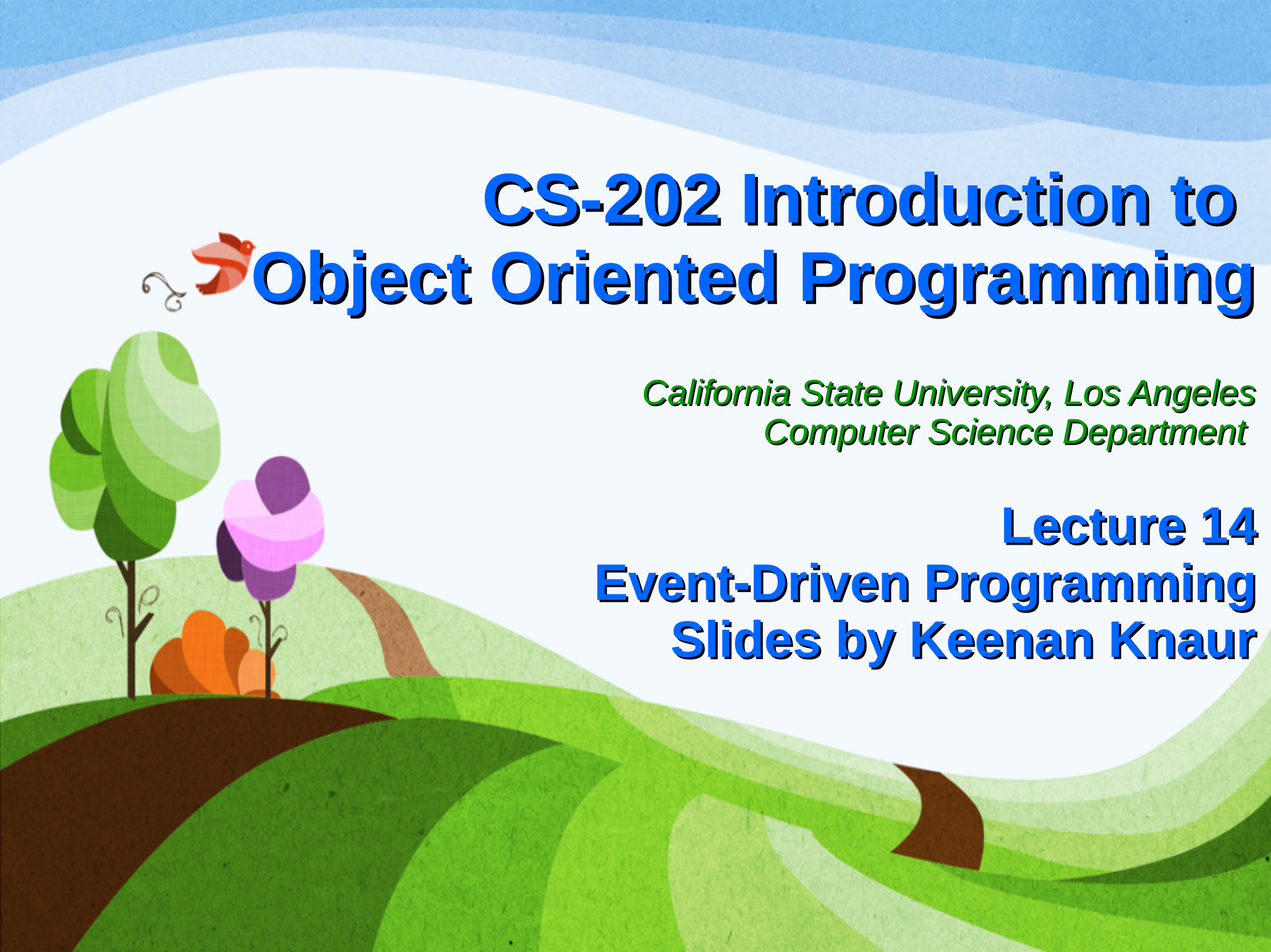
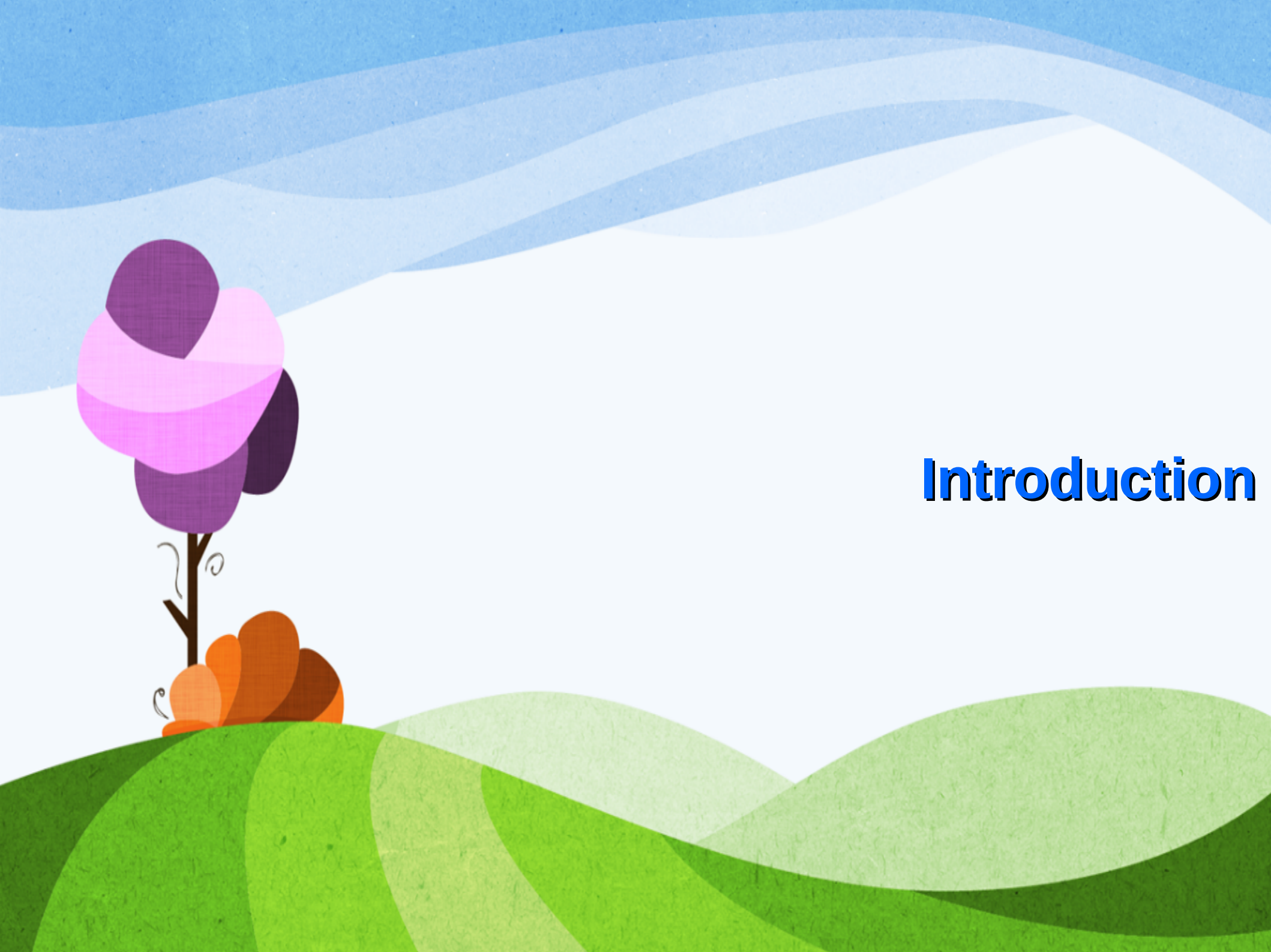


CS-202 Introduction to Object Oriented Programming

California State University, Los Angeles
Computer Science Department

Lecture 14
Event-Driven Programming
Slides by Keenan Knaur


Introduction

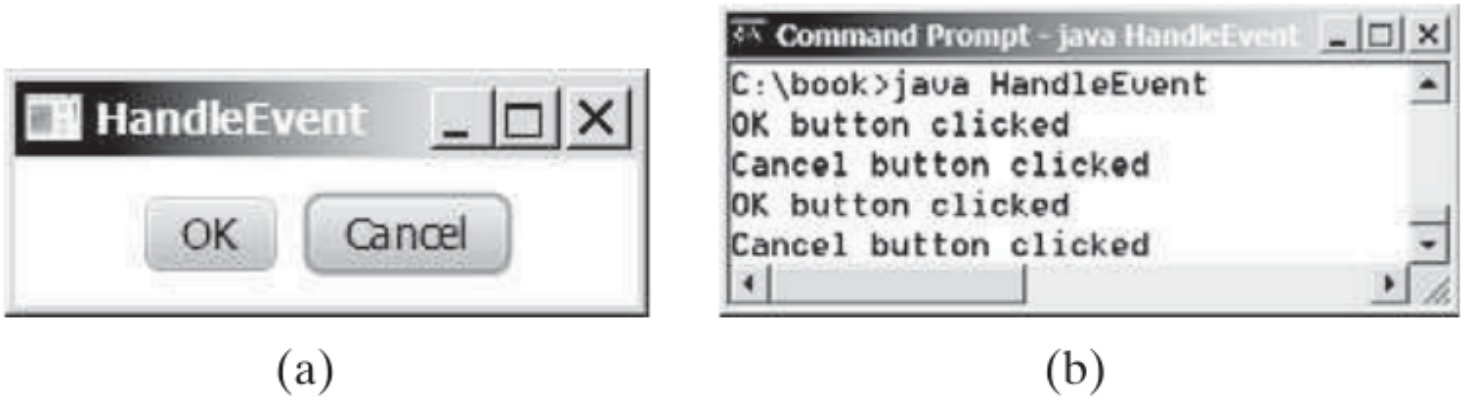
Introduction

- 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.

- 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) (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***

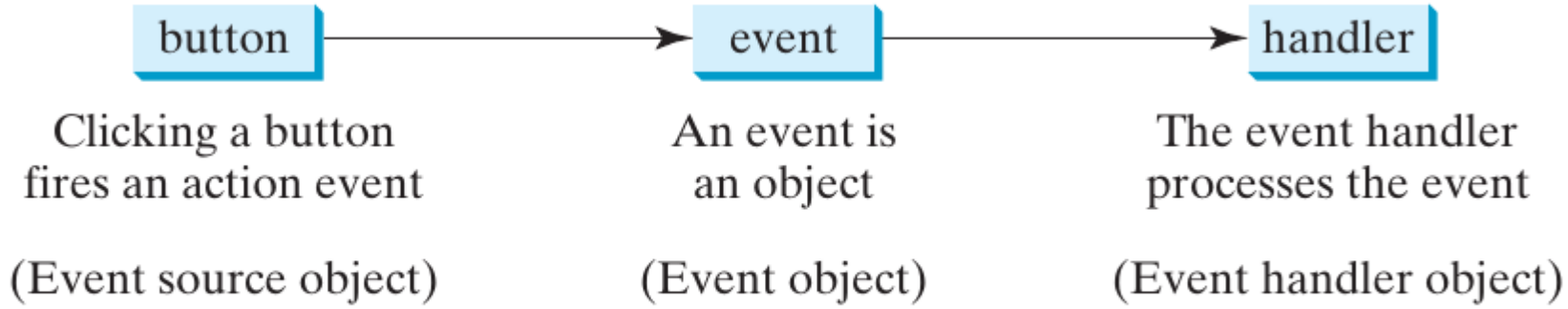
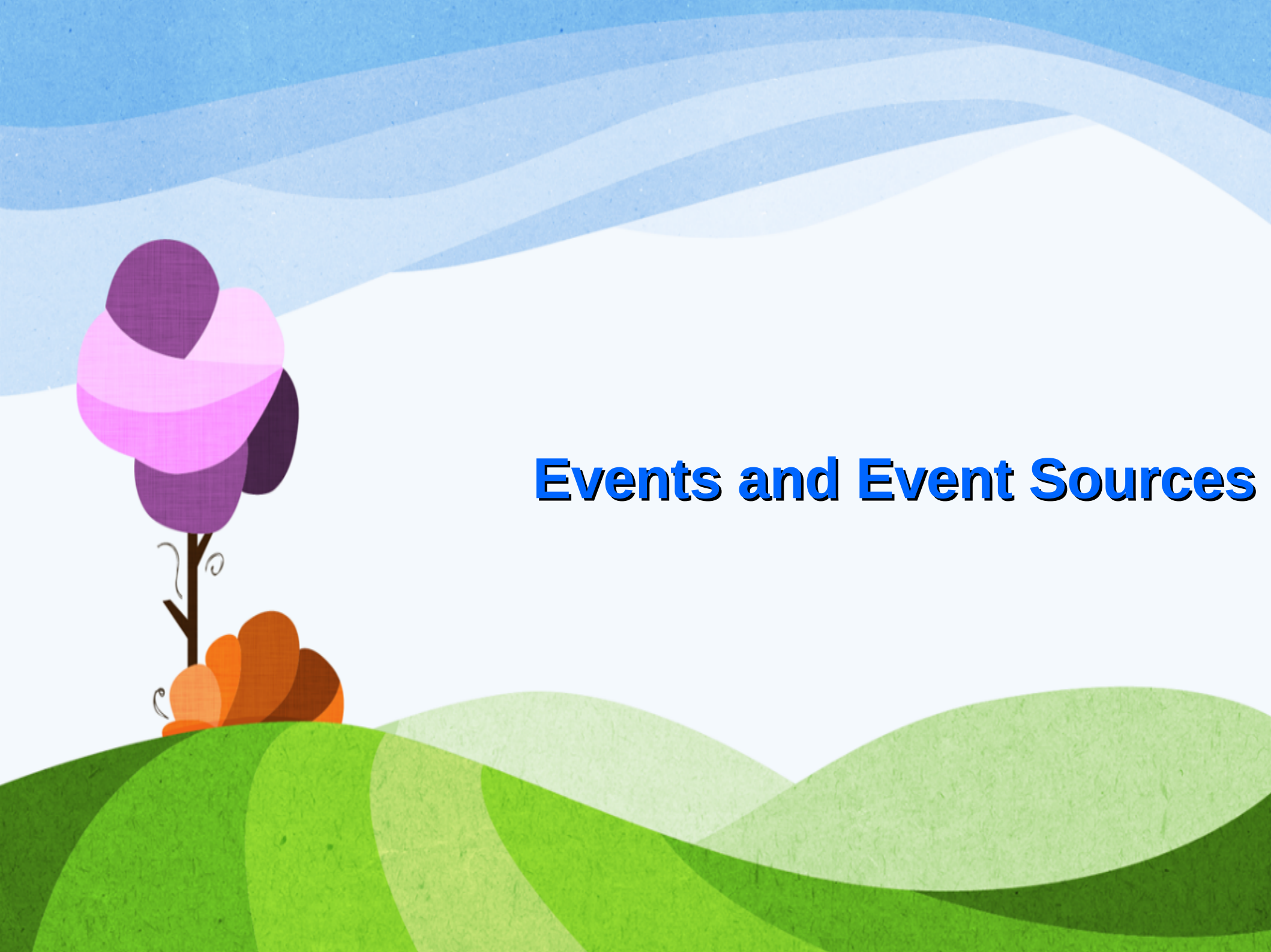


FIGURE 15.3 An event handler processes the event fired from the source object.

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.


Events and Event Sources

Events and Event Sources

- ***event***: object created from an event source, a signal to the program that something has happened.
- ***firing an event***: creating an event and delegating the handler to handle the event.

- a user interacts with a GUI program and the events of the GUI are what drive its execution (this is called ***event-driven programming***)

- 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`**.

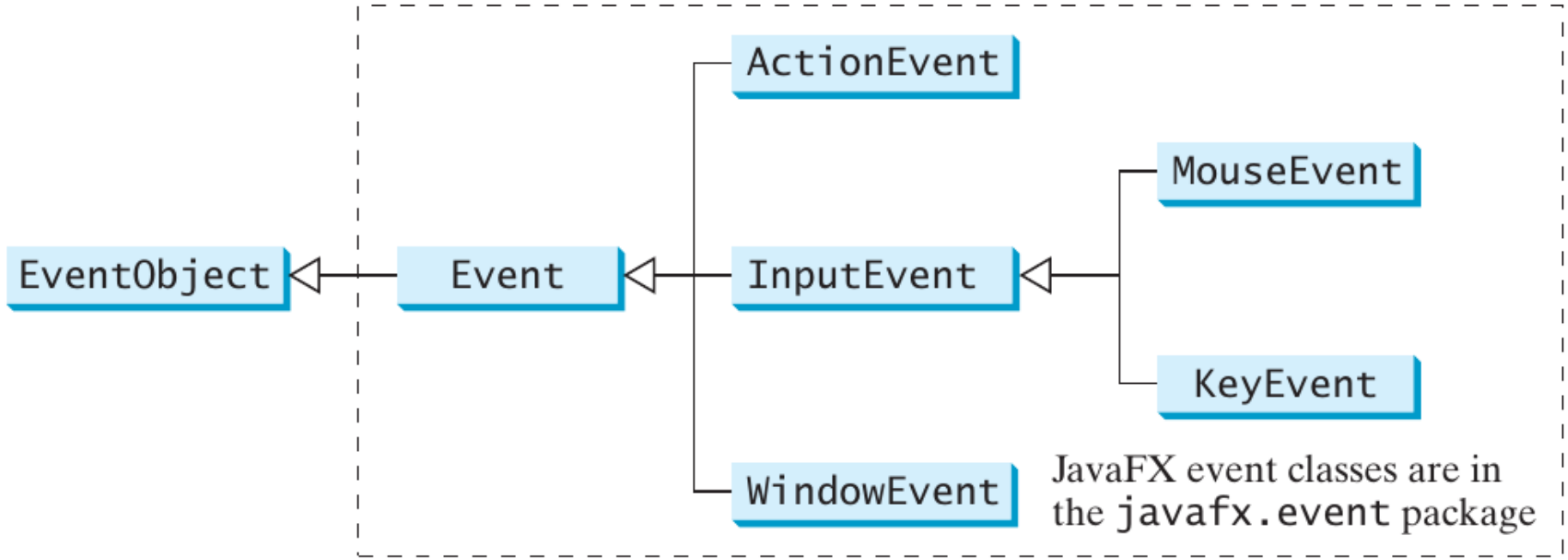


FIGURE 15.4 An event in JavaFX is an object of the `javafx.event.Event` class.

Events and Event Sources

- ***event objects*** contain whatever properties are important to the event

- the **`getSource()`** method of the **`EventObject`** class can be used to identify the source objects.

- 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	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`


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.

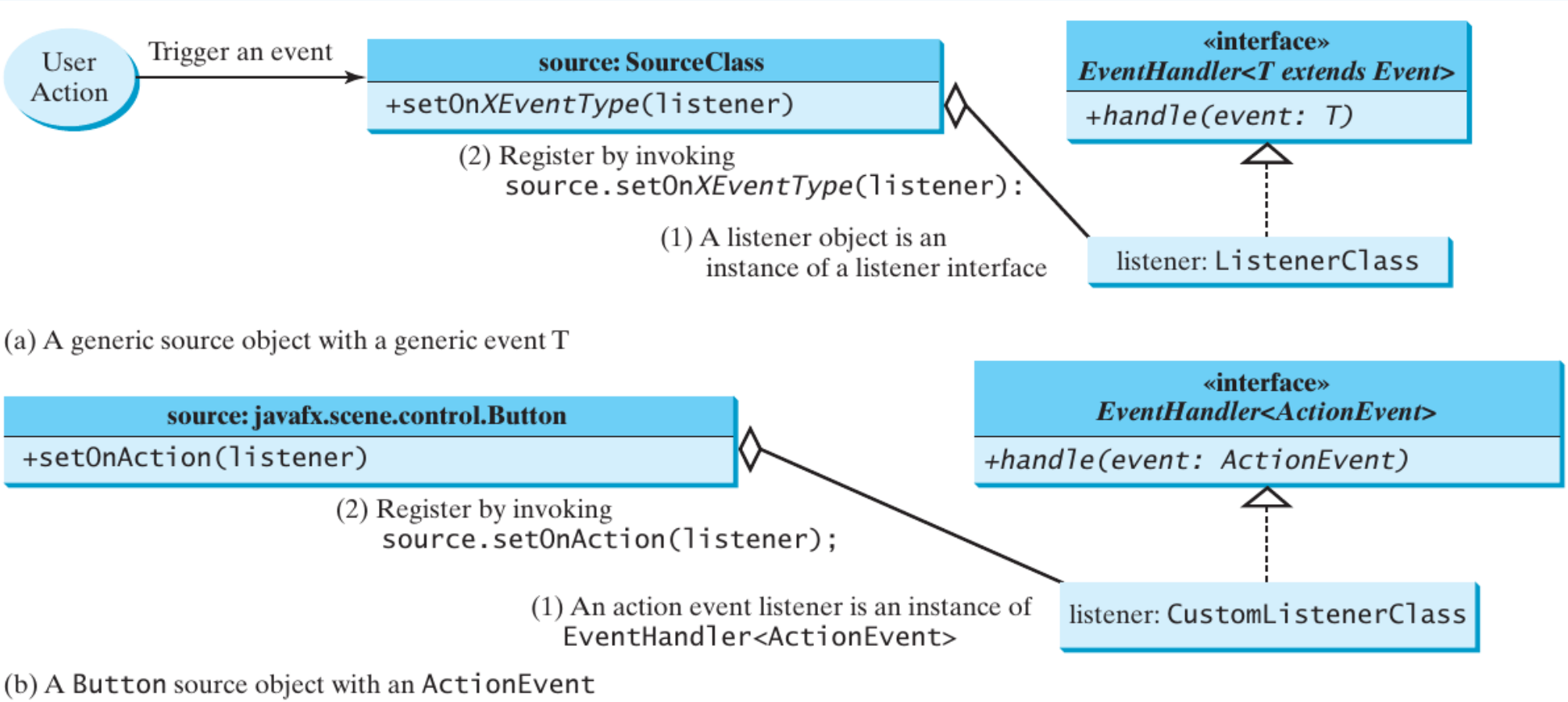


FIGURE 15.5 A listener must be an instance of a listener interface and must be registered with a source object.

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

- Can be used just like regular classes.
- 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 it's own class.
- Compile into a class called **`OuterClassName$InnerclassName.class`**
- 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

- can be defined with a visibility modifier (uses the same rules as previous)
- can be defined as static
  - static inner classes can be accessed using the outer class name
- 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();
```

- 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 EnlargeHandlner
      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

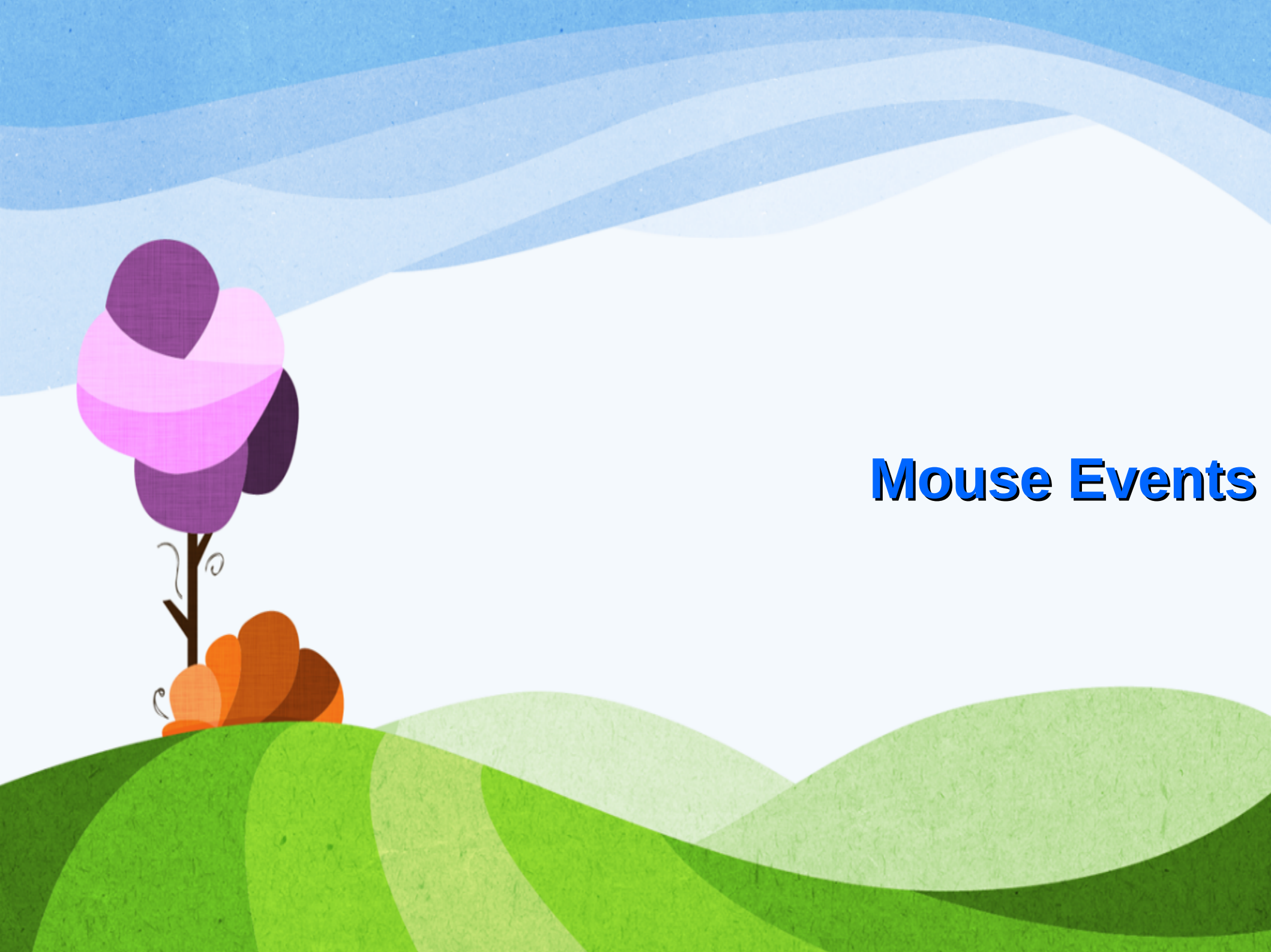
Lambda Expressions

- 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

- 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.

- 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

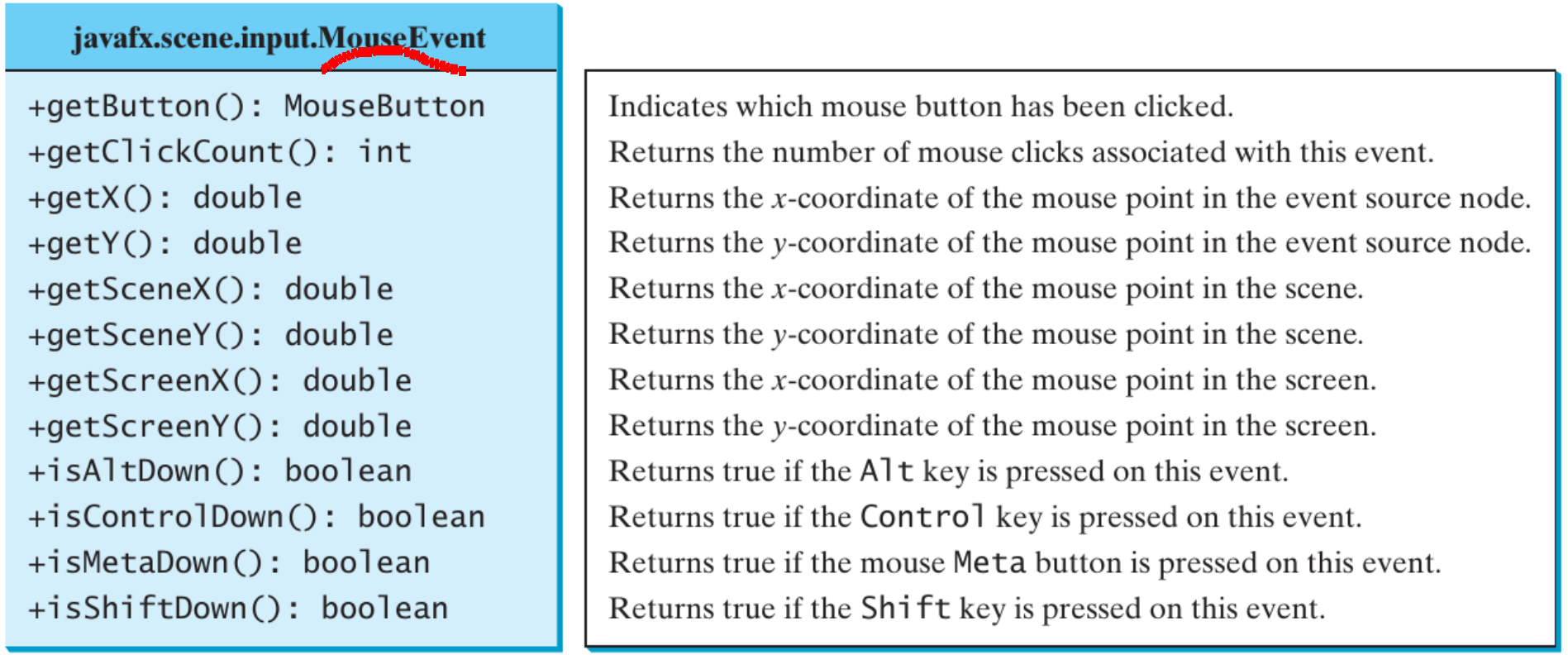


FIGURE 15.10 The `MouseEvent` class encapsulates information for mouse events.

- Also defines four constants **`PRIMARY`**, **`SECONDARY`**, **`MIDDLE`**, and **`NONE`** to indicated 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`	Returns the character associated with the key in this event.
`+getCode(): KeyCode`	Returns the key code associated with the key in this event.
`+getText(): String`	Returns a string describing the key code.
`+isAltDown(): boolean`	Returns true if the `Alt` key is pressed on this event.
`+isControlDown(): boolean`	Returns true if the `Control` key is pressed on this event.
`+isMetaDown(): boolean`	Returns true if the mouse `Meta` button is pressed on this event.
`+isShiftDown(): boolean`	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

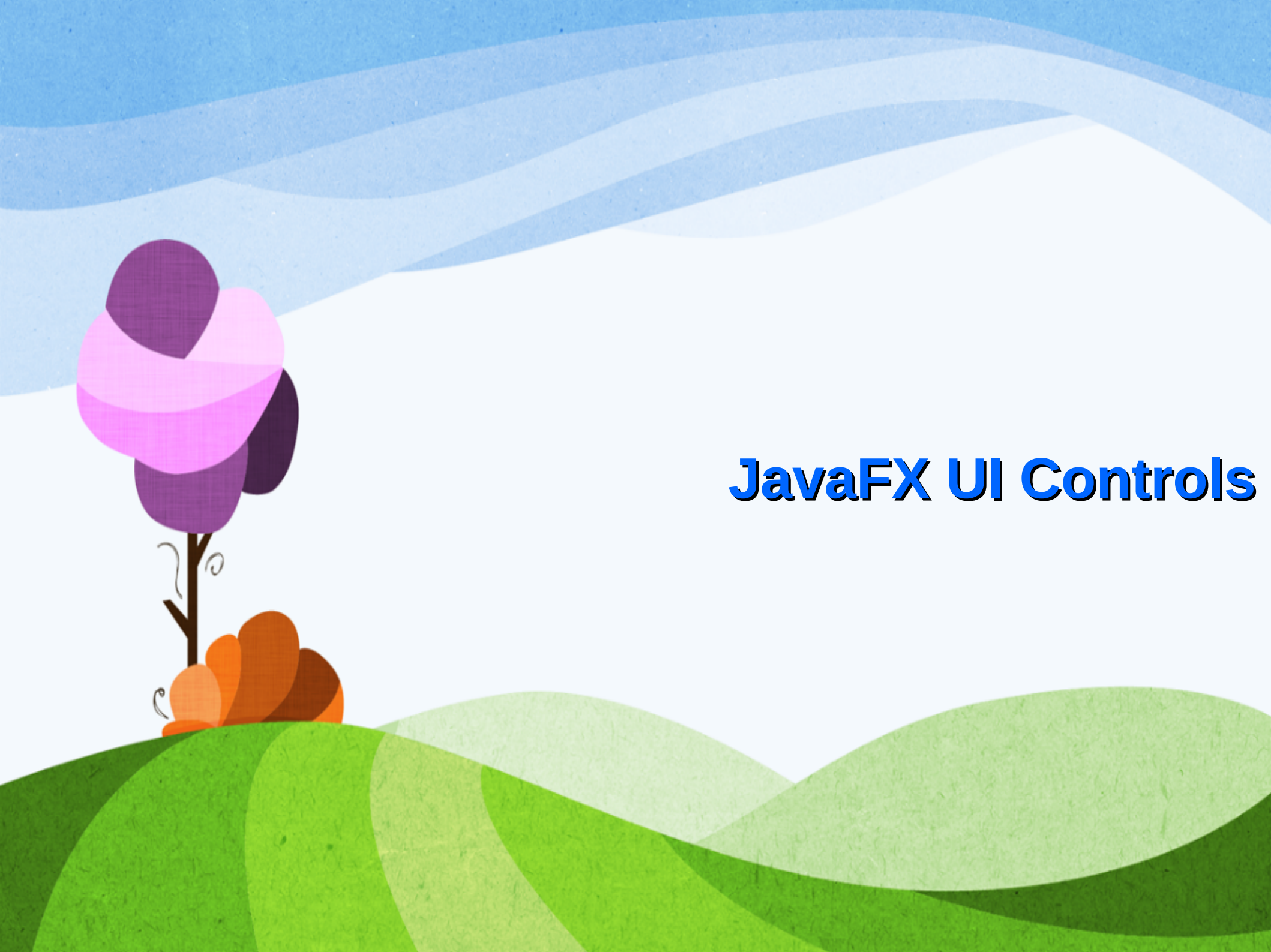
Constant	*Description*	*Constant*	*Description*
HOME	The Home key	CONTROL	The Control key
END	The End key	SHIFT	The Shift key
PAGE_UP	The Page Up key	BACK_SPACE	The Backspace key
PAGE_DOWN	The Page Down key	CAPS	The Caps Lock key
UP	The up-arrow key	NUM_LOCK	The Num Lock key
DOWN	The down-arrow key	ENTER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



JavaFX UI Controls

Common JavaFX UI Controls

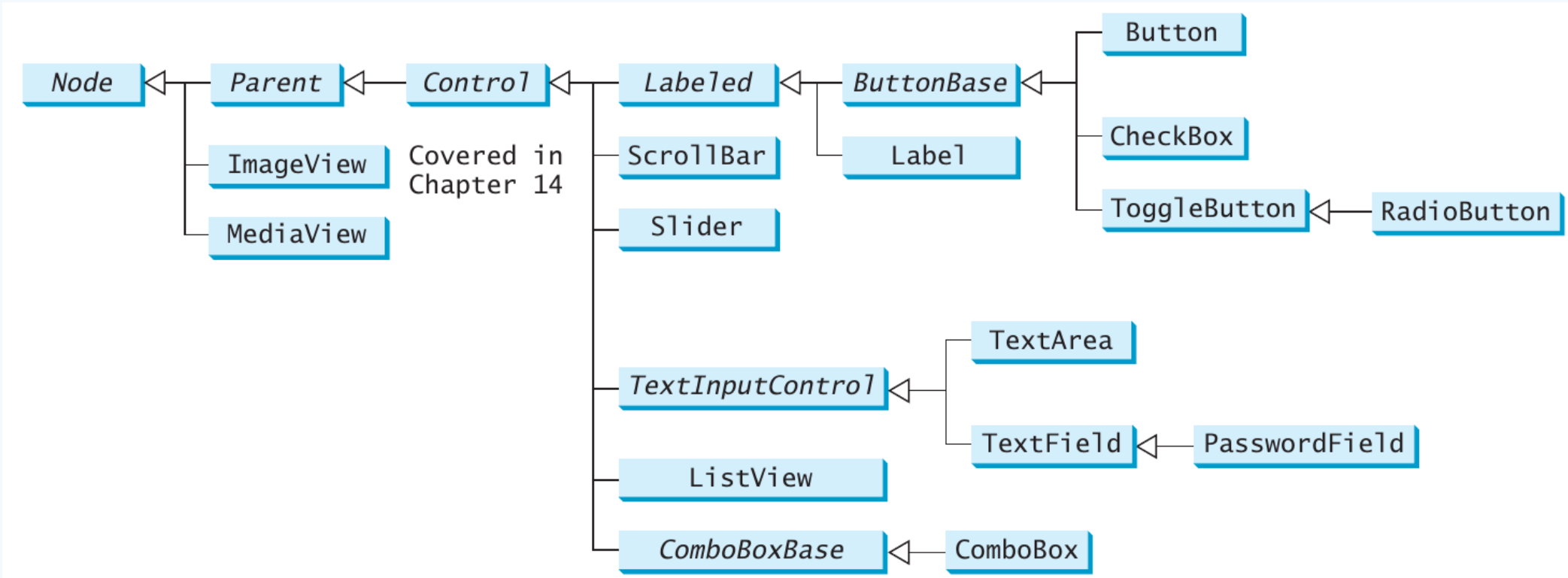


FIGURE 16.1 These UI controls are frequently used to create user interfaces.

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.

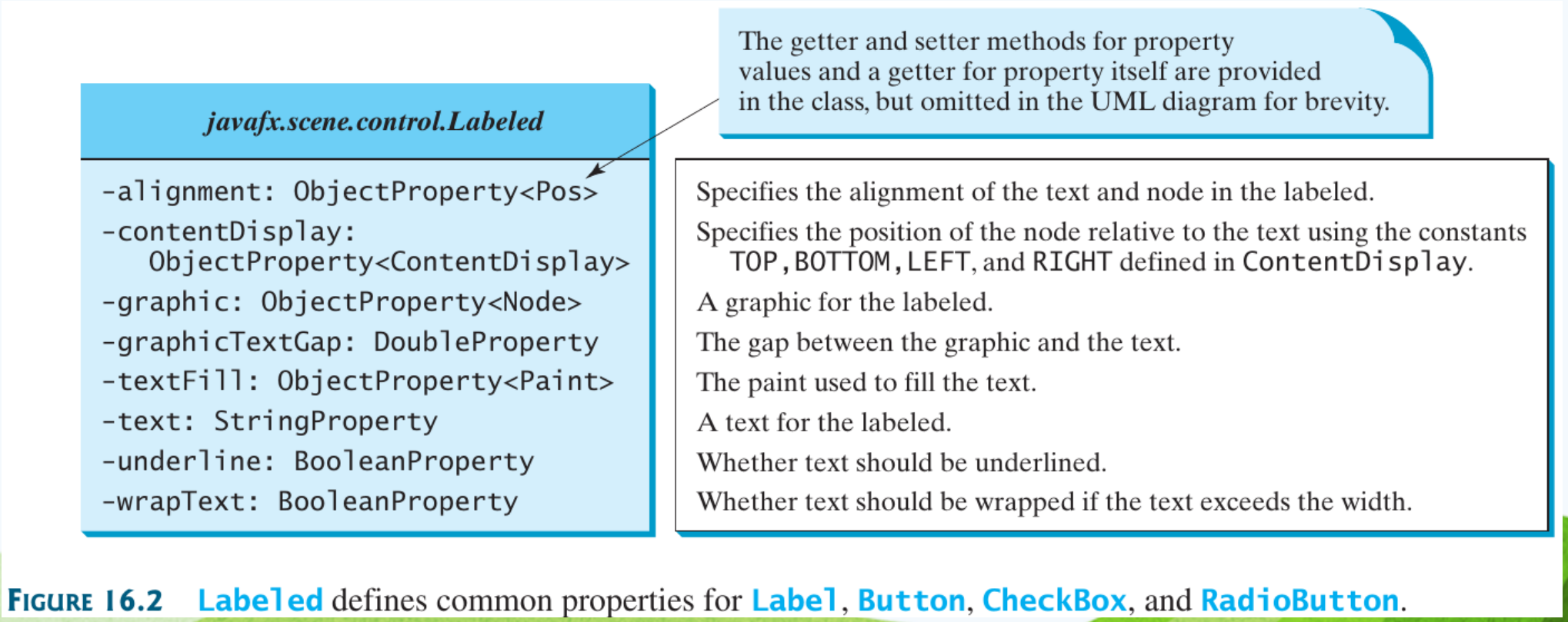


FIGURE 16.2 `Labeled` defines common properties for `Label`, `Button`, `CheckBox`, and `RadioButton`.

Labeled and Label

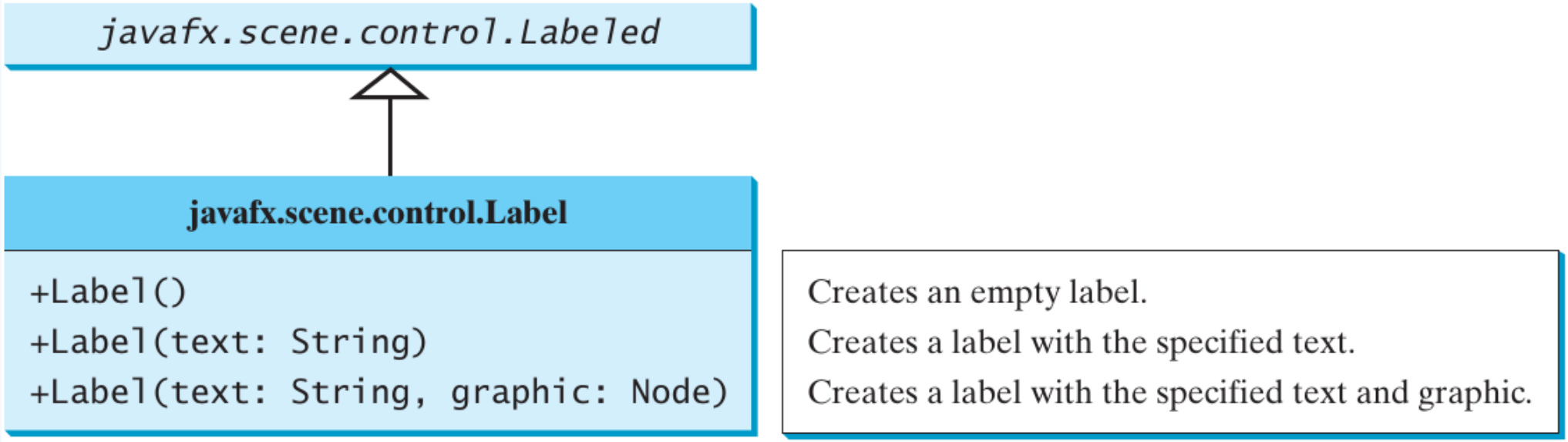


FIGURE 16.3 `Label` is created to display a text or a node, or both.

- See Code: **`LabelWithGraphic.java`**

Button

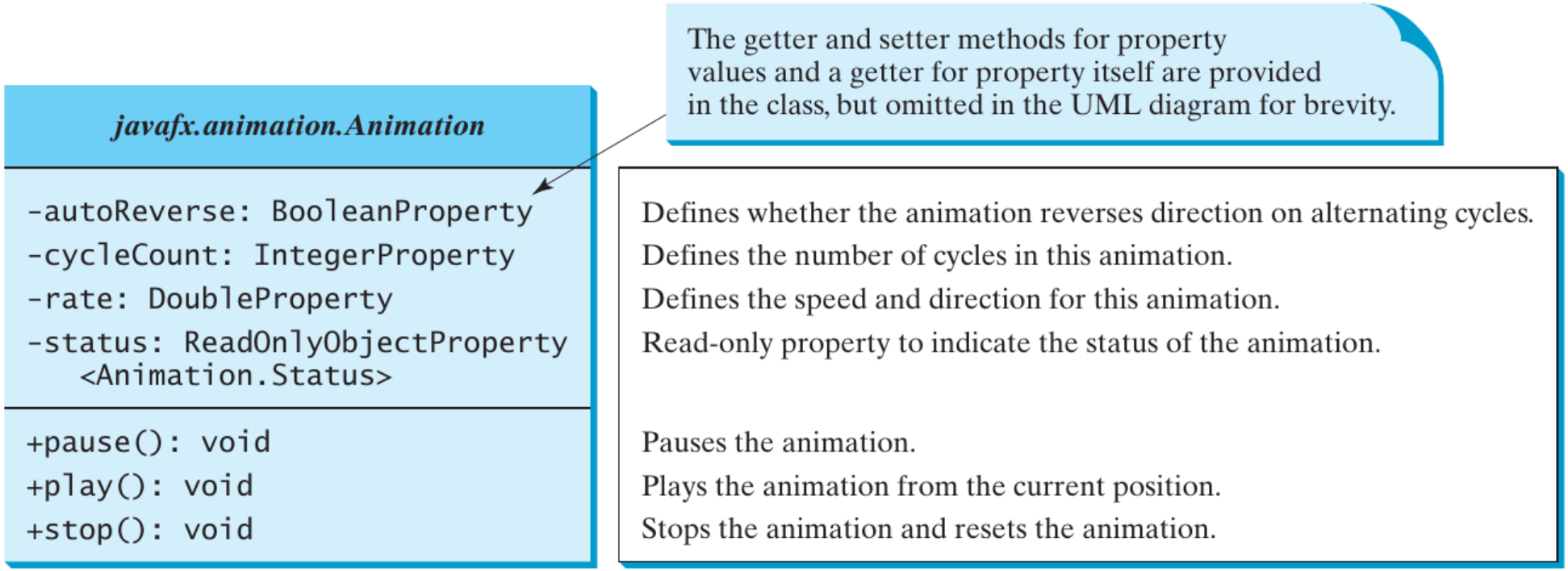


FIGURE 15.15 The abstract `Animation` class is the root class for JavaFX animations.

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`**

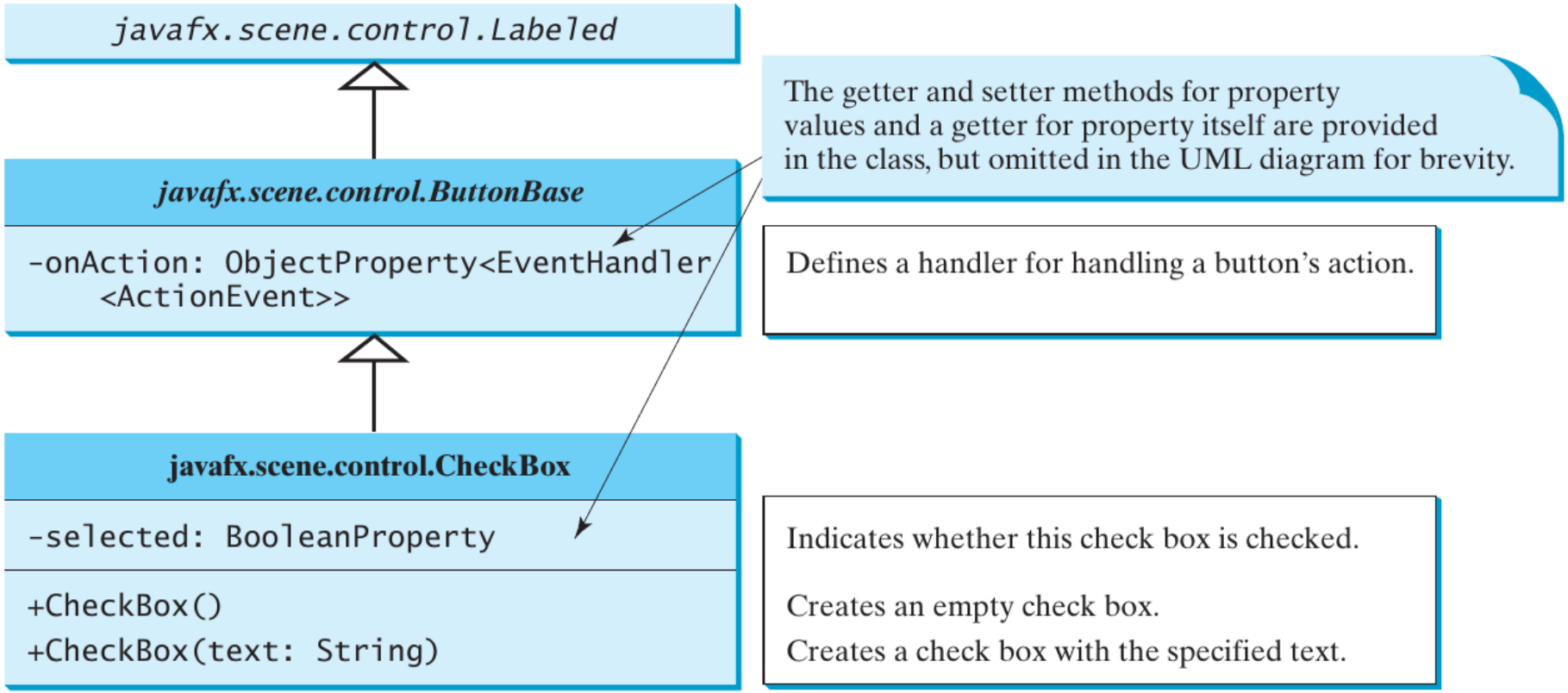


FIGURE 16.7 CheckBox contains the properties inherited from ButtonBase and Labeled.

RadioButton

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

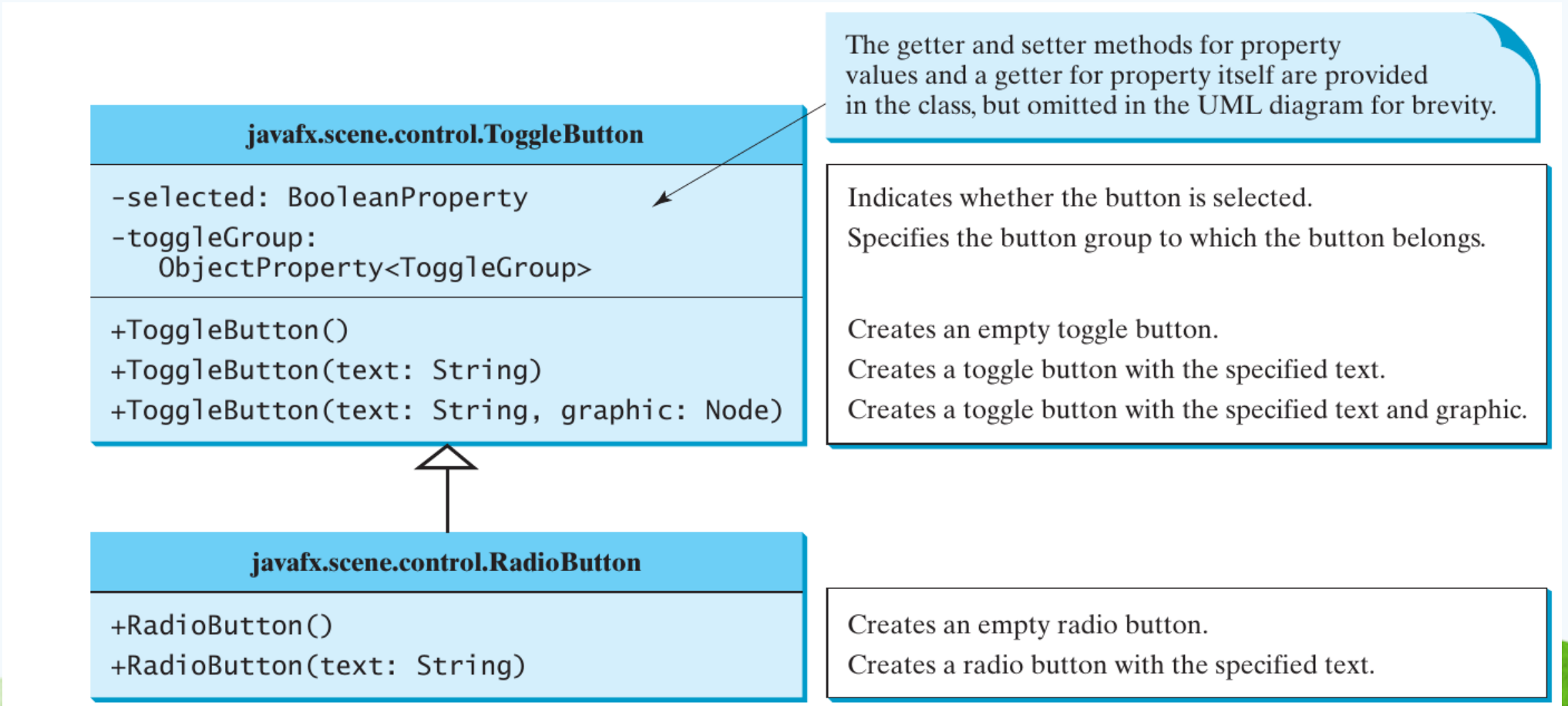


FIGURE 16.9 `ToggleButton` and `RadioButton` are specialized buttons for making selections.

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,));
```

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

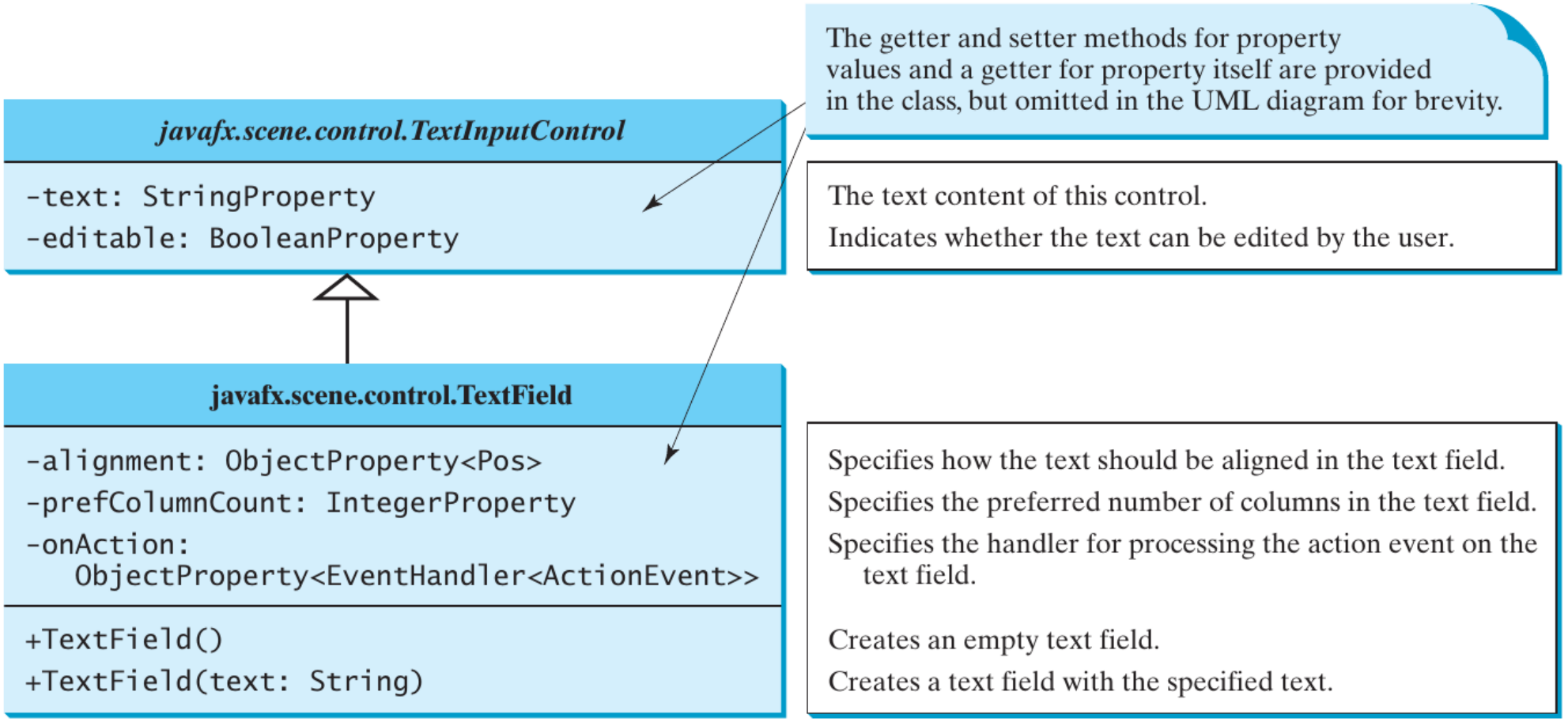


FIGURE 16.11 `TextField` enables the user to enter or display a string.

- See **`TextFieldDemo.java`**

TextArea

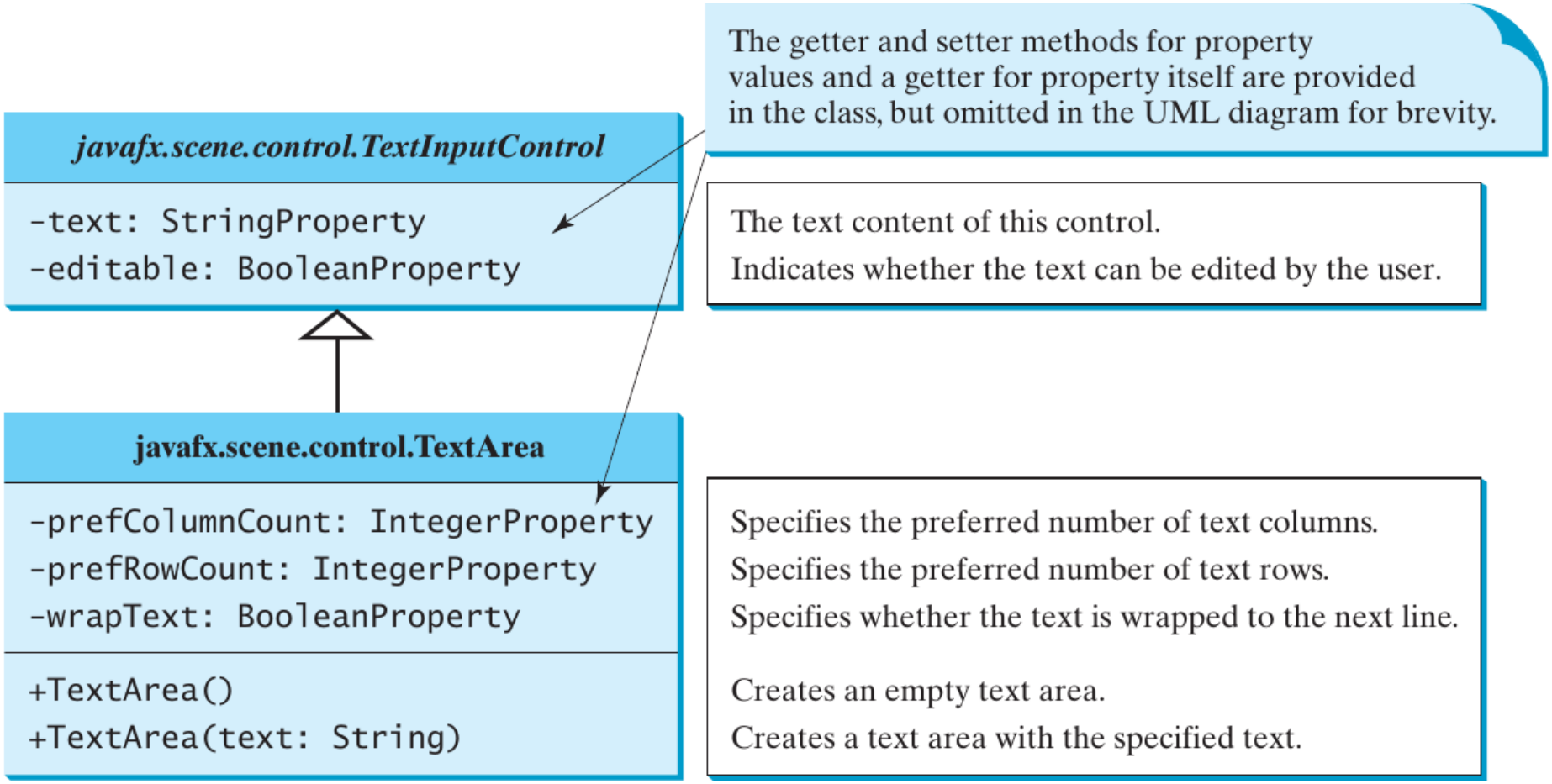


FIGURE 16.13 `TextArea` enables the user to enter or display multiple lines of characters.

- 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`**

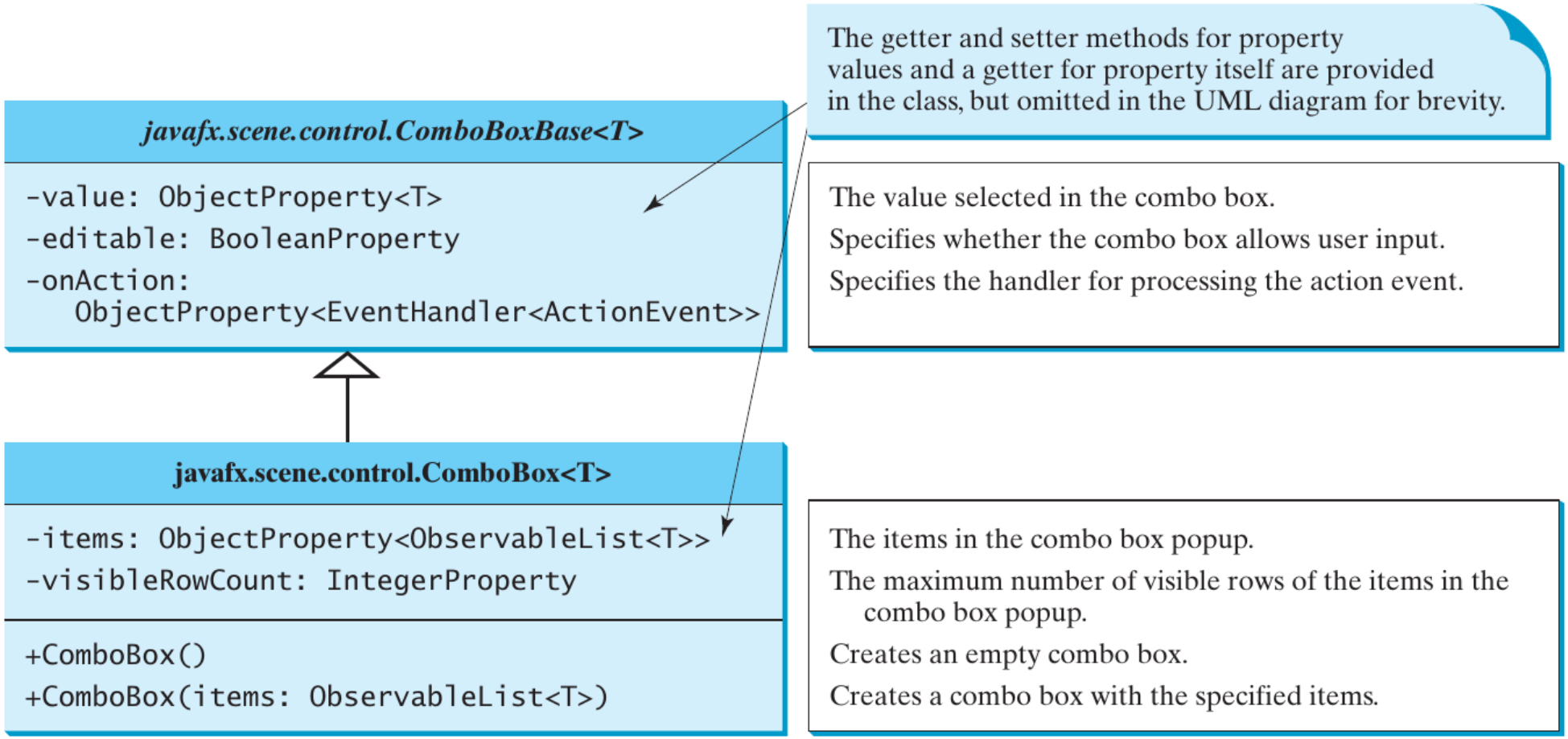


FIGURE 16.16 ComboBox enables the user to select an item from a list of items.

ListView

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

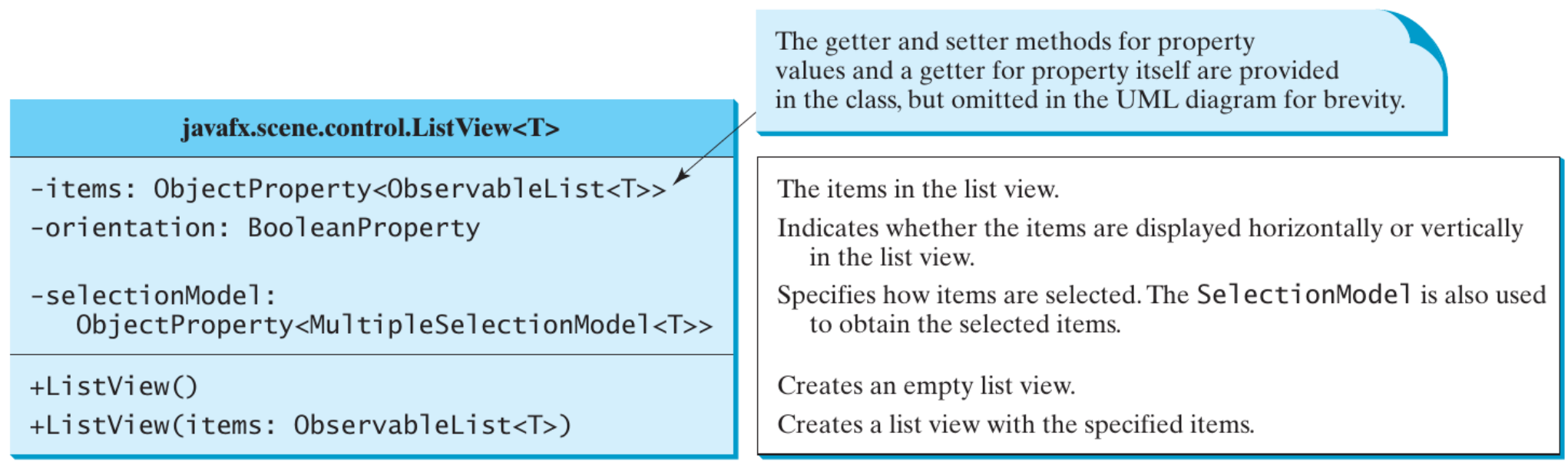


FIGURE 16.18 `ListView` enables the user to select one or multiple items from a list of items.

ScrollBar

- See Code: **`ScrollBarDemo.java`**

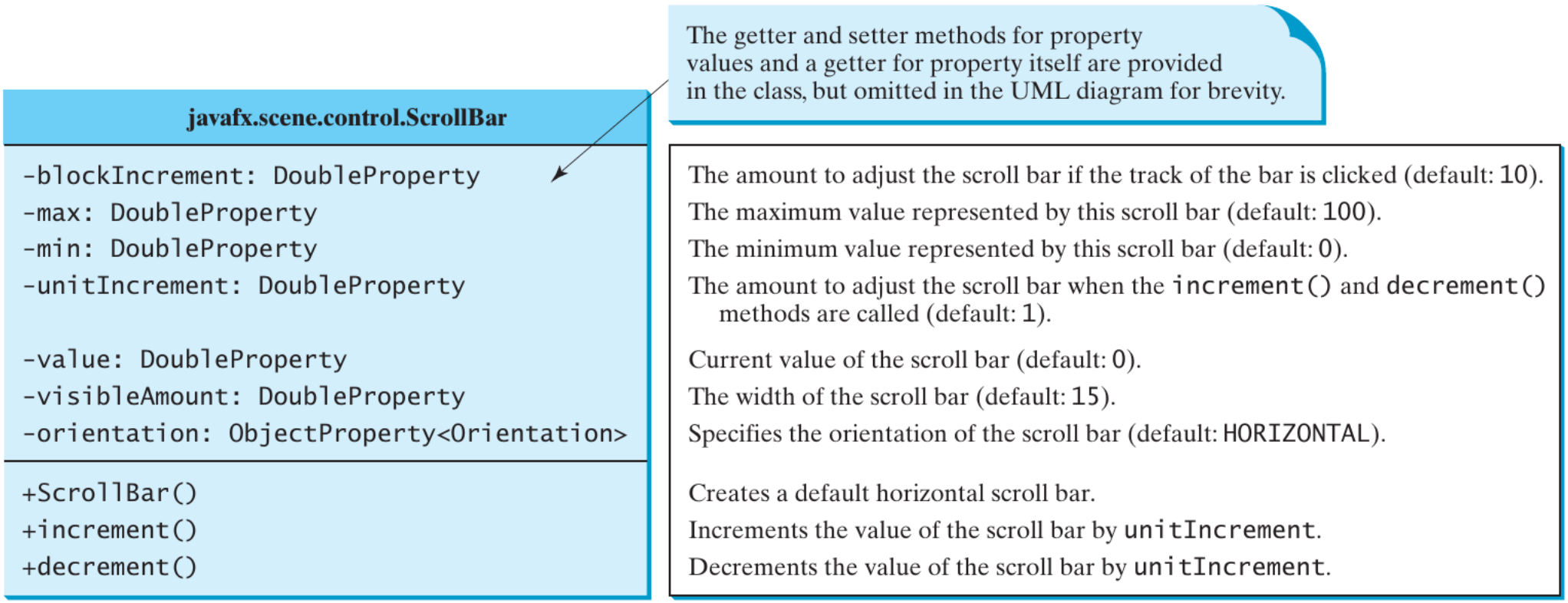


FIGURE 16.22 `ScrollBar` enables the user to select from a range of values.

Slider

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

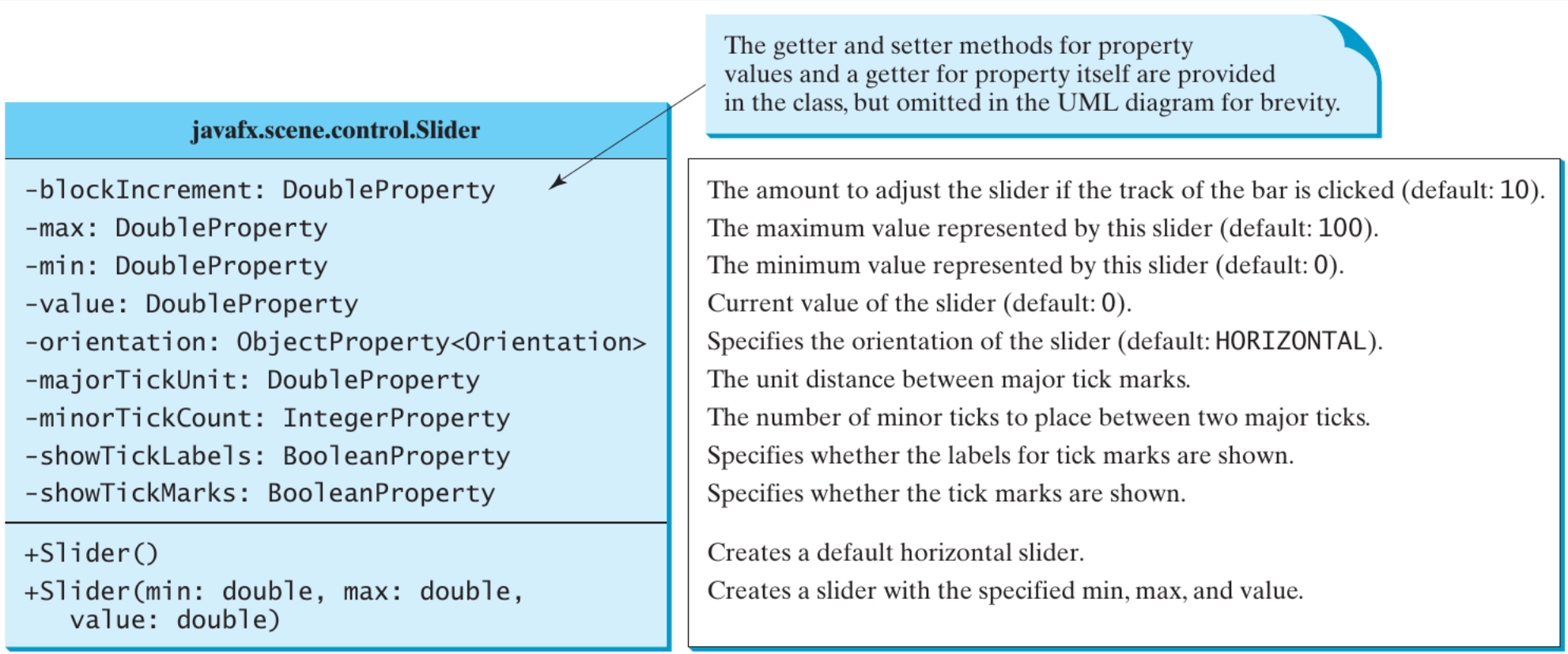


FIGURE 16.25 `Slider` enables the user to select from a range of values.