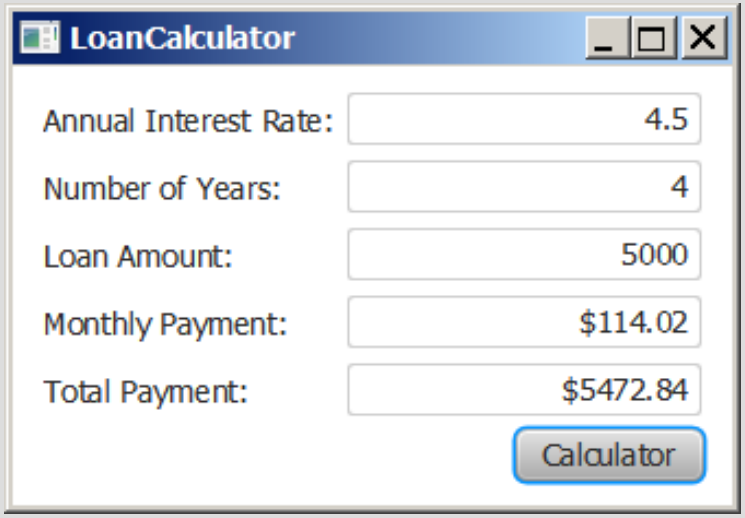
COIS2240 Lecture11

Motivations

Suppose you want to write a GUI program that lets the user enter a loan amount, annual interest rate, and number of years and click the *Compute Payment* button to obtain the monthly payment and total payment. How do you accomplish the task? You have to use *event-driven programming* to write the code to respond to the button-clicking event.



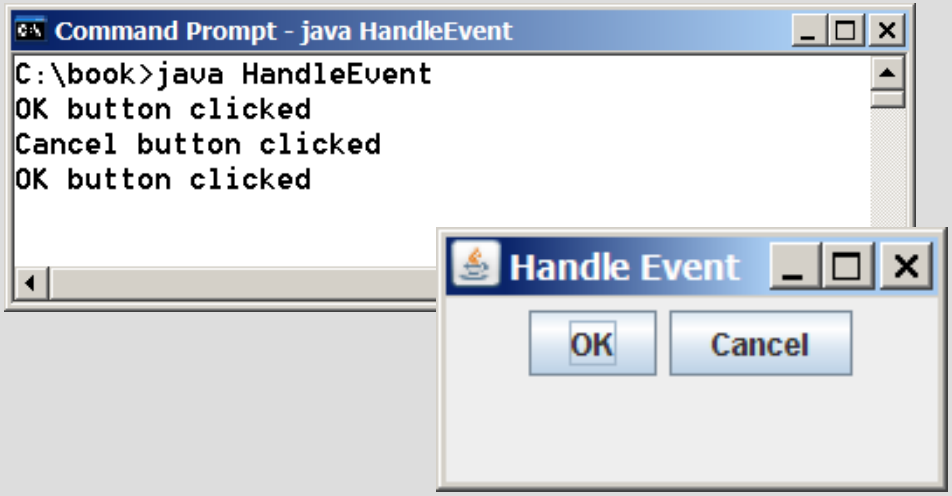
See code www.cs.armstrong.edu/liang/intro11e/html/LoanCalculator.html

Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order.
- In event-driven programming, code is executed upon activation of events.

Taste of Event-Driven Programming

The example displays a button in the frame. A message is displayed on the console when a button is clicked.

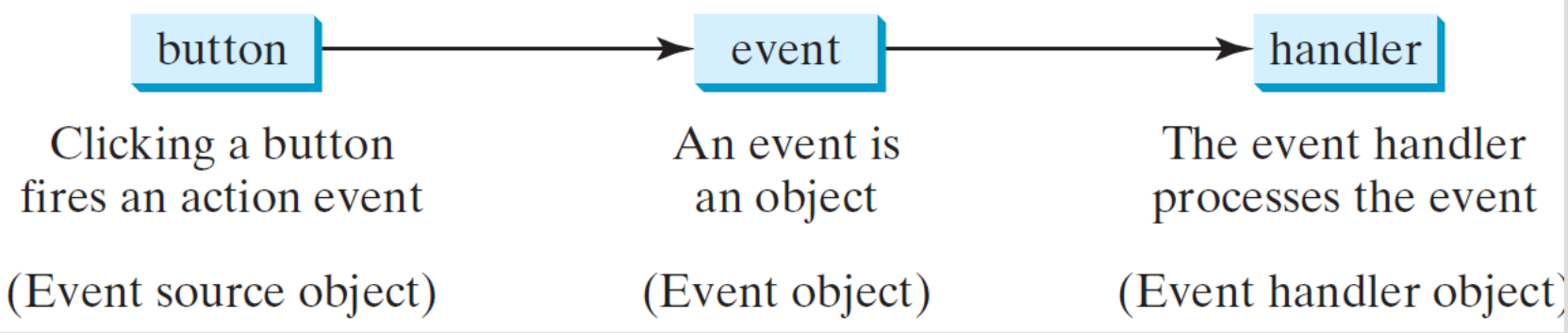


See code: http://www.cs.armstrong.edu/liang/intro11e/html/HandleEvent.html

Handling GUI Events

Source object (e.g., button)

Listener object (handler) contains a method for processing the event.

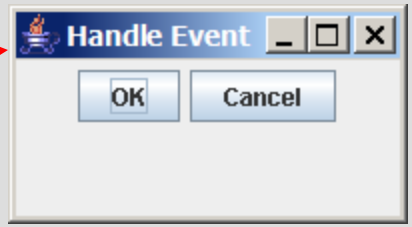


Trace Execution

```
public class HandleEvent extends Application {
  public void start(Stage primaryStage) {
    …
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    …
    primaryStage.show(); // Display the stage
  }
}

class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

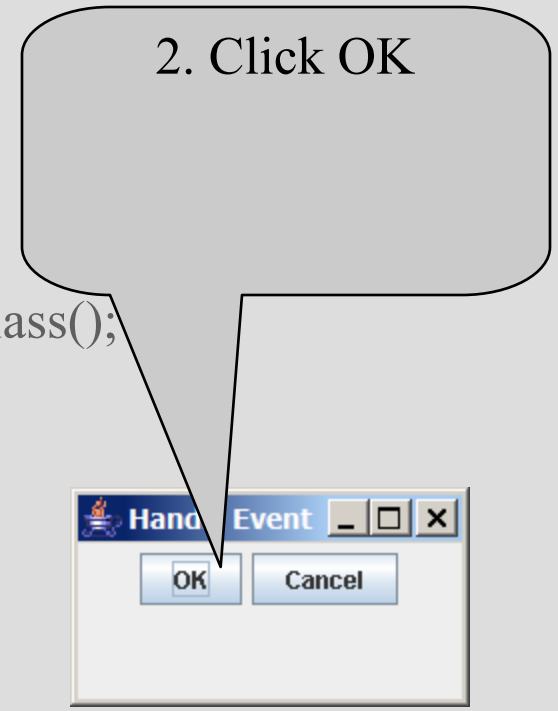
1. Start from the main method to create a window and display it



Trace Execution

```
public class HandleEvent extends Application {
  public void start(Stage primaryStage) {
    …
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    …
    primaryStage.show(); // Display the stage
  }
}

class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

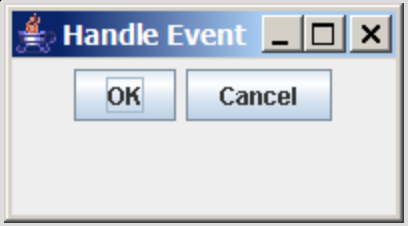


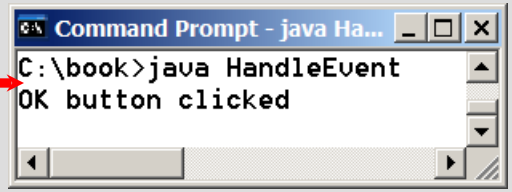
Trace Execution

```
public class HandleEvent extends Application {
  public void start(Stage primaryStage) {
    …
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    …
    primaryStage.show(); // Display the stage
  }
}

class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

3. The JVM invokes the listener's handle method





Events

```
class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

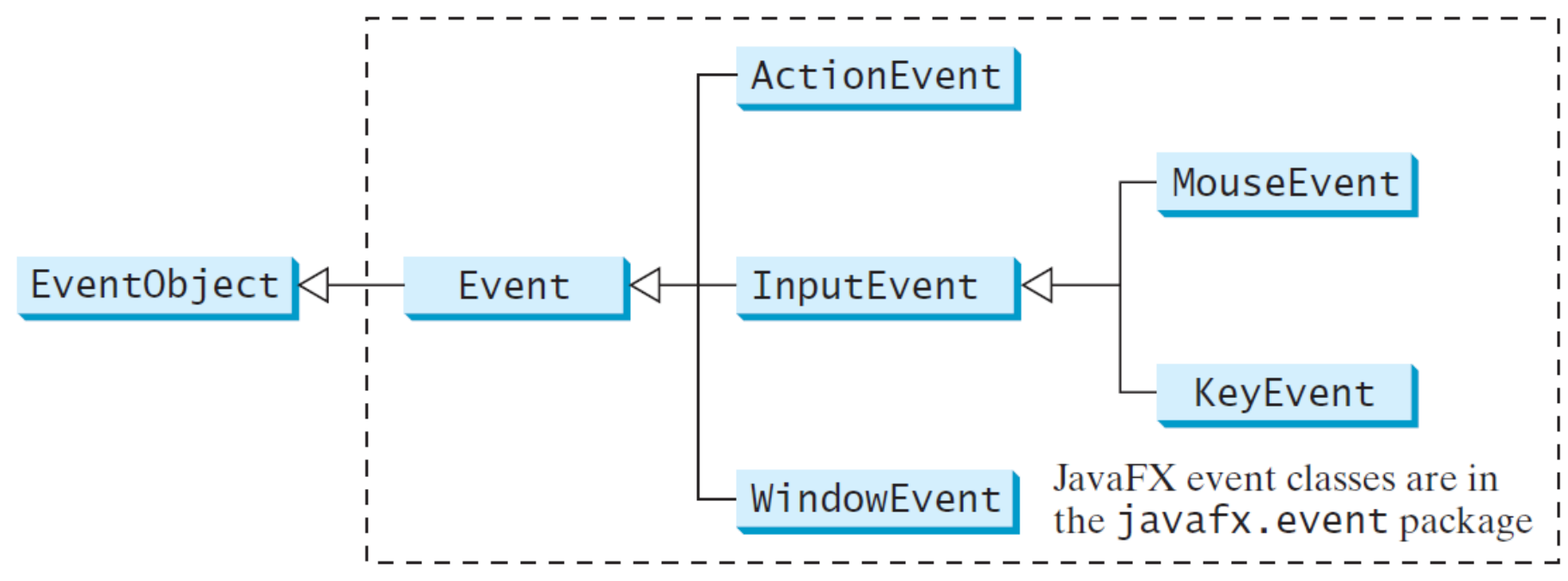
- EventHandler objects must be an instance of the **EventHandler<T extends Event>** interface. This interface defines the common behaviour for all handlers. **<T extends Event>** denotes that **T** is a generic type that is a subtype of **Event.**
- An *event* can be defined as a type of signal to the program that something has happened.
- The event is generated by external user actions such as mouse movements, mouse clicks, or keystrokes.

Events

```
class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

- EventHandler objects must be an instance of the **EventHandler<T extends Event>** interface. This interface defines the common behavior for all handlers. **<T extends Event>** denotes that **T** is a generic type that is a subtype of **Event.**
- Remember generic types? According to Java Docs - A generic type is a generic class or interface that is parameterized over types.

Event Classes



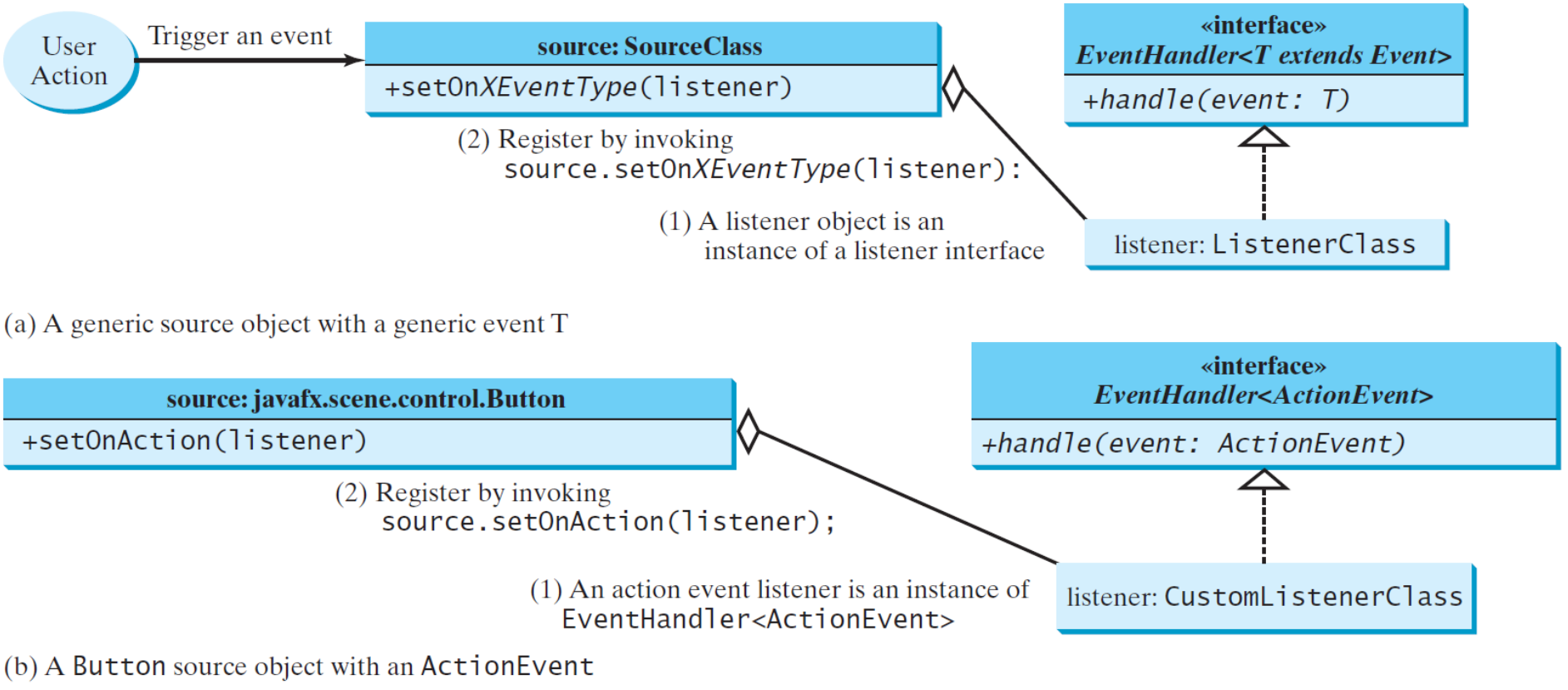
Event Information

An event object contains whatever properties are pertinent to the event. You can identify the source object of the event using the getSource() instance method in the EventObject class. The subclasses of EventObject deal with special types of events, such as button actions, window events, mouse movements, and keystrokes. The Table in the next slide lists external user actions, source objects, and event types generated.

Selected User Actions and Handlers

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>)`

The Delegation Model

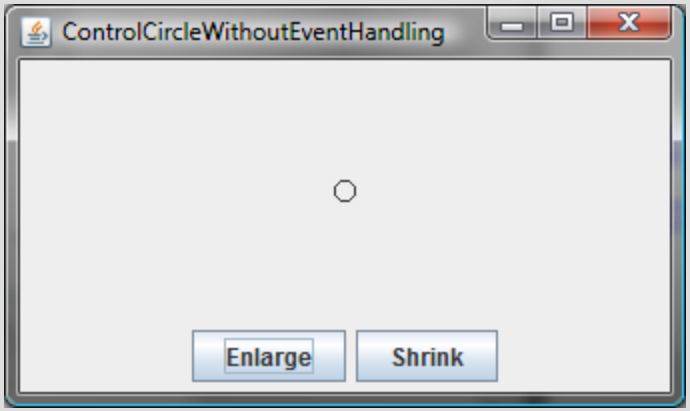


The Delegation Model: Example

```
Button btOK = new Button("OK");
OKHandlerClass handler = new OKHandlerClass();
btOK.setOnAction(handler);
```

Example: First Version for ControlCircle (no listeners)

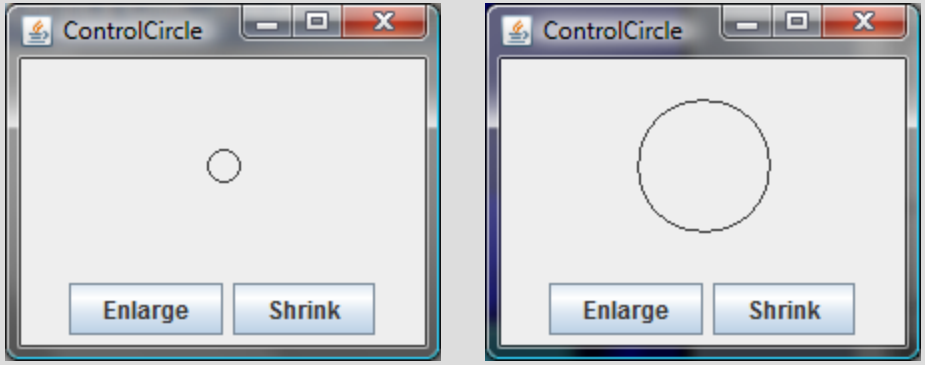
Now let us consider to write a program that uses two buttons to control the size of a circle.



See code: www.cs.armstrong.edu/liang/intro11e/html/ControlCircleWithoutEventHandling.html

Example: Second Version for ControlCircle (with listener for Enlarge)

Now let us consider to write a program that uses two buttons to control the size of a circle.



See code:
http://www.cs.armstrong.edu/liang/intro11e/html/ControlCircle.html

Inner Class Listeners

A listener class is designed specifically to create a listener object for a GUI component (e.g., a button). It will not be shared by other applications. So, it is appropriate to define the listener class inside the frame class as an inner class.

Inner Classes

Inner class: A class is a member of another class.

Advantages: In some applications, you can use an inner class to make programs simple.

An inner class can reference the data and methods defined in the outer class in which it nests, so you do not need to pass the reference of the outer class to the constructor of the inner class.

Inner Classes, cont.

```
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)

Inner Classes (cont.)

- An inner class can be declared public, protected, or private subject to the same visibility rules applied to a member of the class.
- An inner class can be declared static. A static inner class can be accessed using the outer class name. A static inner class cannot access nonstatic members of the outer class

Anonymous Inner Classes

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit extends or implements clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class 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().

Anonymous Inner Classes (cont.)

Inner class listeners can be shortened using anonymous inner classes. An *anonymous inner class* is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step. An anonymous inner class is declared as follows:

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface
  // Other methods if necessary
}
```

Anonymous Inner Classes (cont.)

```
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

See example code:

http://www.cs.armstrong.edu/liang/intro11e/html/AnonymousHandlerDemo.html

The `MouseEvent` Class

javafx.scene.input.MouseEvent	
`+getButton(): MouseButton`	Indicates which mouse button has been clicked.
`+getClickCount(): int`	Returns the number of mouse clicks associated with this event.
`+getX(): double`	Returns the *x*-coordinate of the mouse point in the event source node.
`+getY(): double`	Returns the *y*-coordinate of the mouse point in the event source node.
`+getSceneX(): double`	Returns the *x*-coordinate of the mouse point in the scene.
`+getSceneY(): double`	Returns the *y*-coordinate of the mouse point in the scene.
`+getScreenX(): double`	Returns the *x*-coordinate of the mouse point in the screen.
`+getScreenY(): double`	Returns the *y*-coordinate of the mouse point in the screen.
`+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.

See code:
http://www.cs.armstrong.edu/liang/intro11e/html/MouseEventDemo.html

The `KeyEvent` Class

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.

See code:
www.cs.armstrong.edu/liang/intro11e/html/KeyEventDemo.html

The `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

Example: Control Circle with Mouse and Key

See code: www.cs.armstrong.edu/liang/intro11e/html/ControlCircleWithMouseAndKey.html

Listeners for Observable Objects

You can add a listener to process a value change in an observable object.

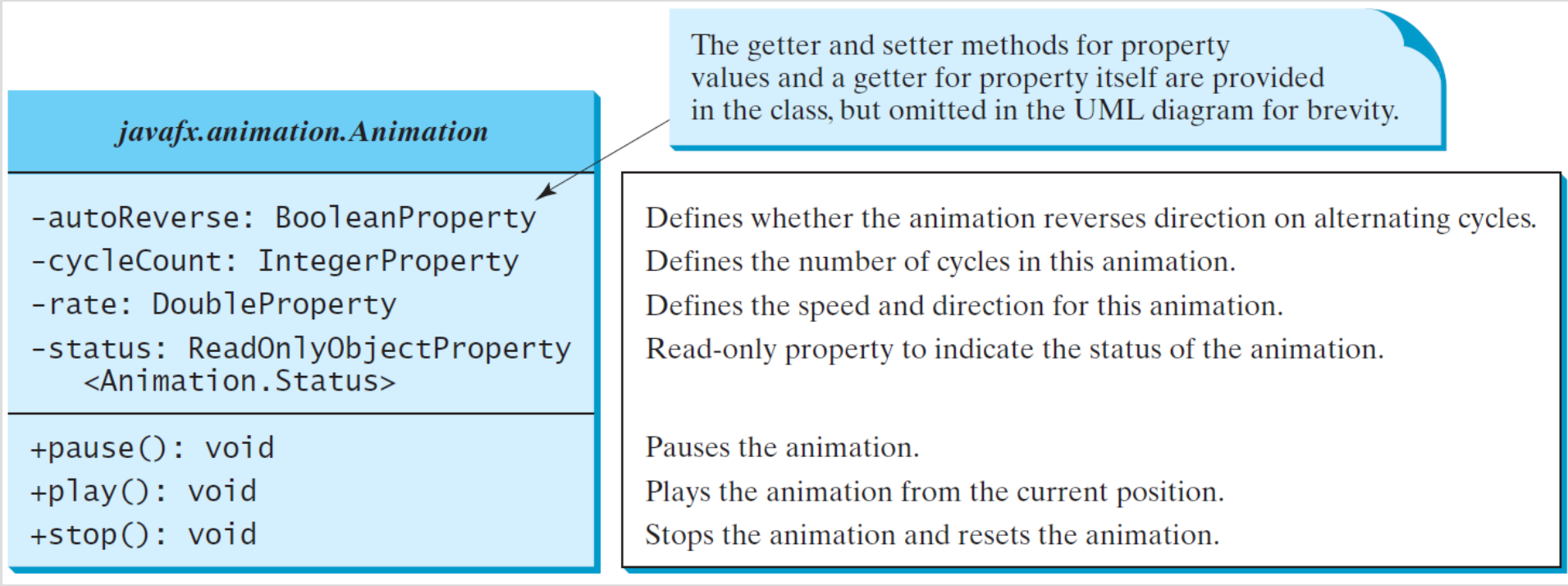
An instance of **Observable** is known as an *observable object*, which contains the **addListener(InvalidationListener listener)** method for adding a listener. Once the value is changed in the property, a listener is notified. The listener class should implement the **InvalidationListener** interface, which uses the **invalidated(Observable o)** method to handle the property value change. Every binding property is an instance of **Observable**.

See code: www.cs.armstrong.edu/liang/intro11e/html/ObservablePropertyDemo.html

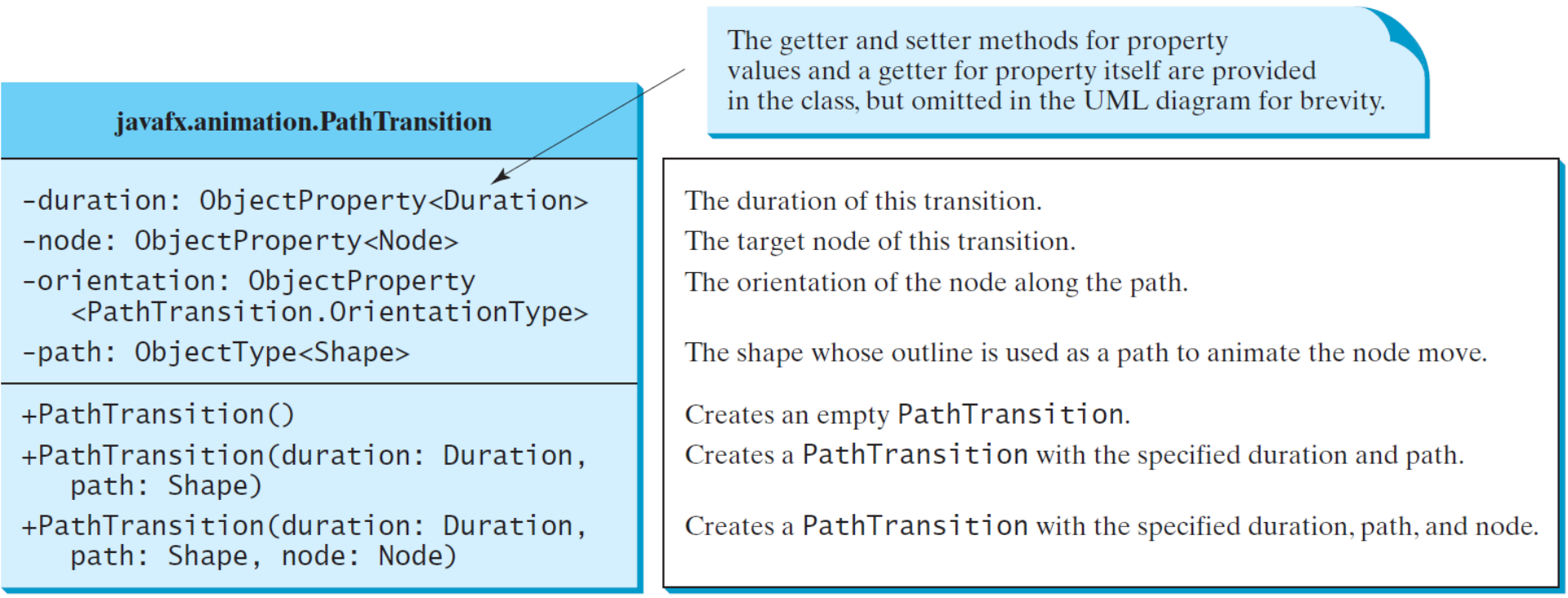
See code: www.cs.armstrong.edu/liang/intro11e/html/DisplayResizableClock.html

Animation

JavaFX provides the **Animation** class with the core functionality for all animations.



PathTransition

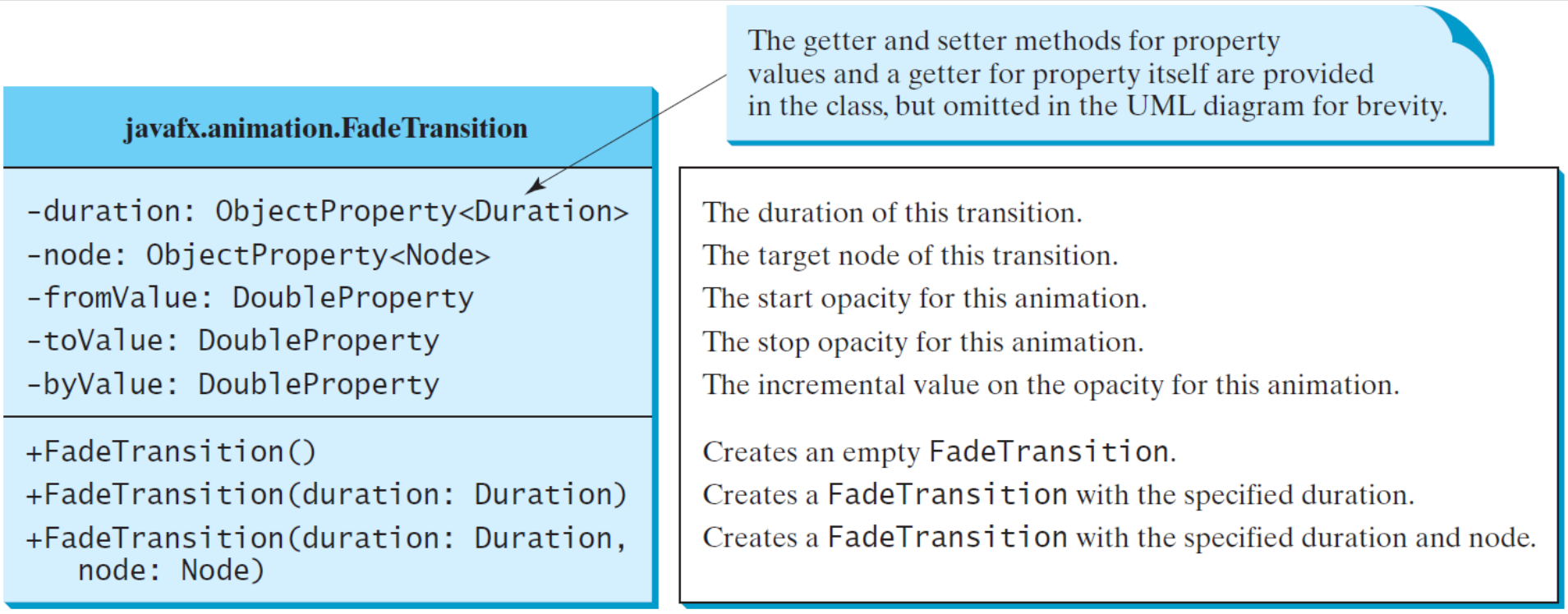


See code examples:

www.cs.armstrong.edu/liang/intro11e/html/PathTransitionDemo.html

FadeTransition

The **FadeTransition** class animates the change of the opacity in a node over a given time.



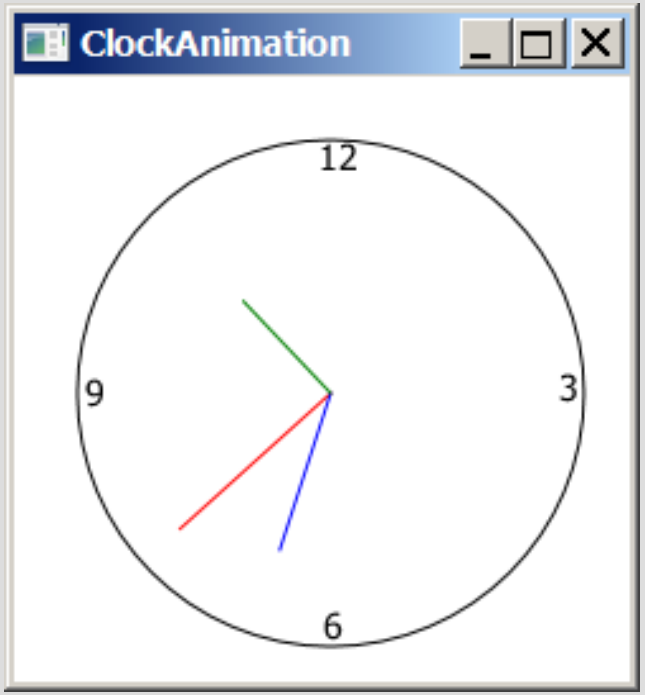
See code:
www.cs.armstrong.edu/liang/intro11e/html/FadeTransitionDemo.html

Timeline

PathTransition and **FadeTransition** define specialized animations. The **Timeline** class can be used to program any animation using one or more **KeyFrame**s. Each **KeyFrame** is executed sequentially at a specified time interval. **Timeline** inherits from **Animation**.

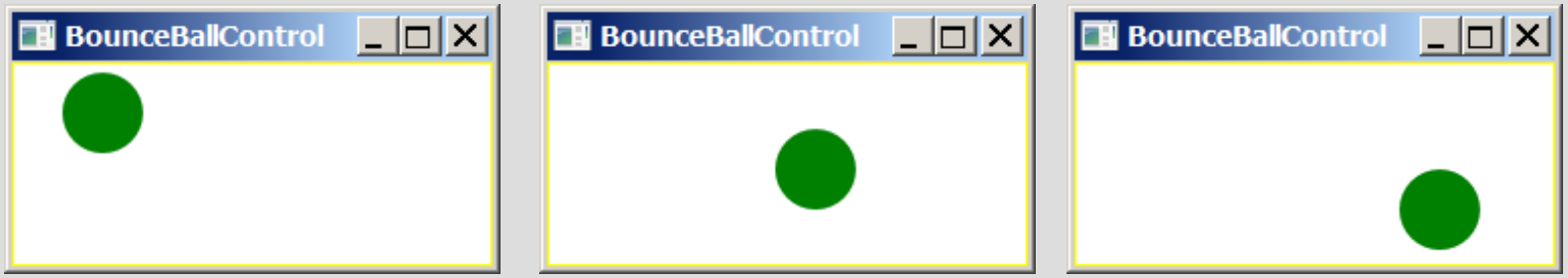
See code:
www.cs.armstrong.edu/liang/intro11e/html/TimelineDemo.html

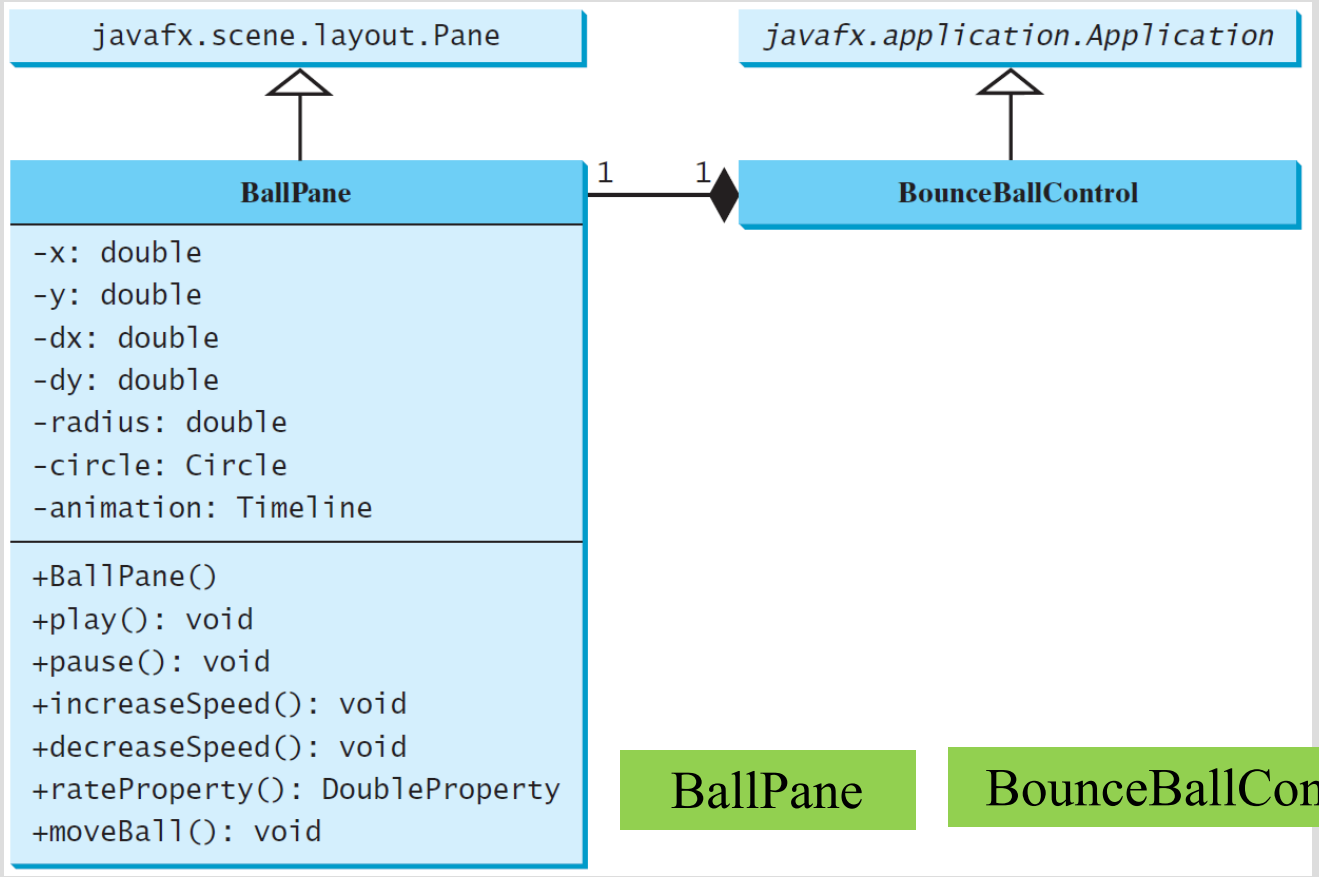
Clock Animation



See code:
www.cs.armstrong.edu/liang/intro11e/html/ClockAnimation.html

Case Study: Bouncing Ball



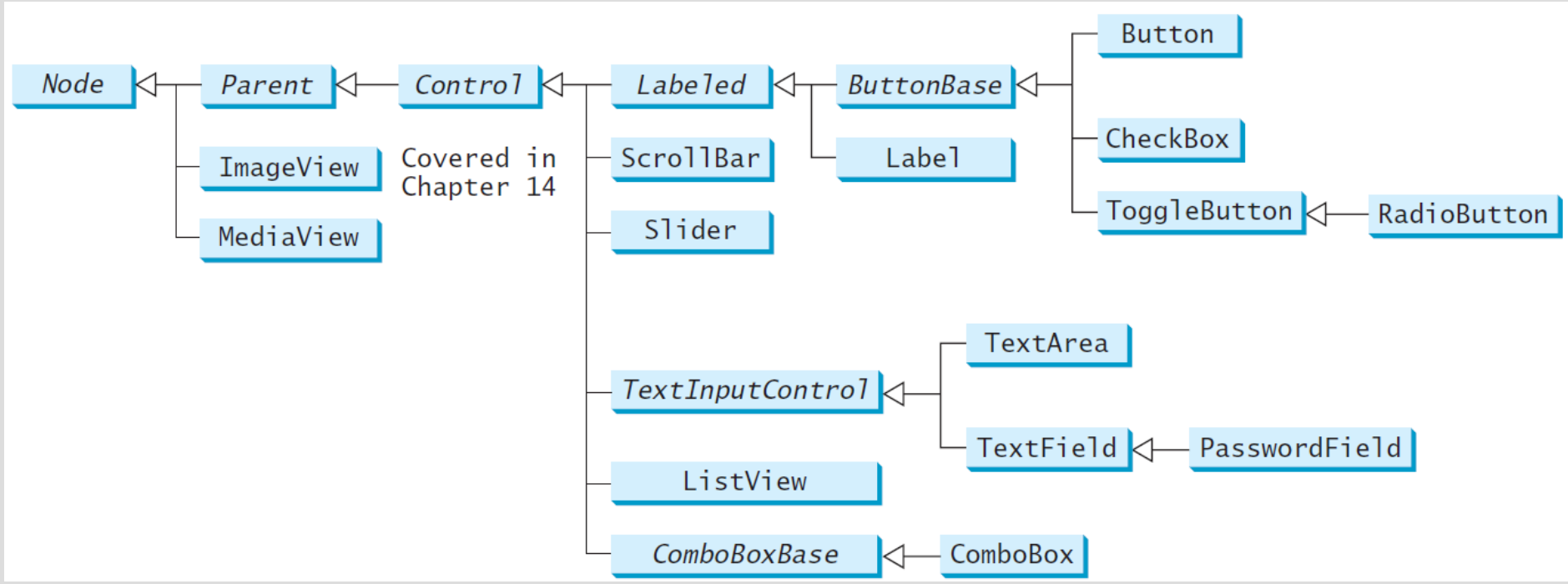


BallPane

BounceBallControl

Run

Frequently Used UI Controls



Throughout this course, the prefixes **lbl**, **bt**, **chk**, **rb**, **tf**, **pf**, **ta**, **cbo**, **lv**, **scb**, **sld**, and **mp** are used to name reference variables for **Label**, **Button**, **CheckBox**, **RadioButton**, **TextField**, **PasswordField**, **TextArea**, **ComboBox**, **ListView**, **ScrollBar**, **Slider**, and **MediaPlayer**.