

Chapter 15 Event-Driven Programming

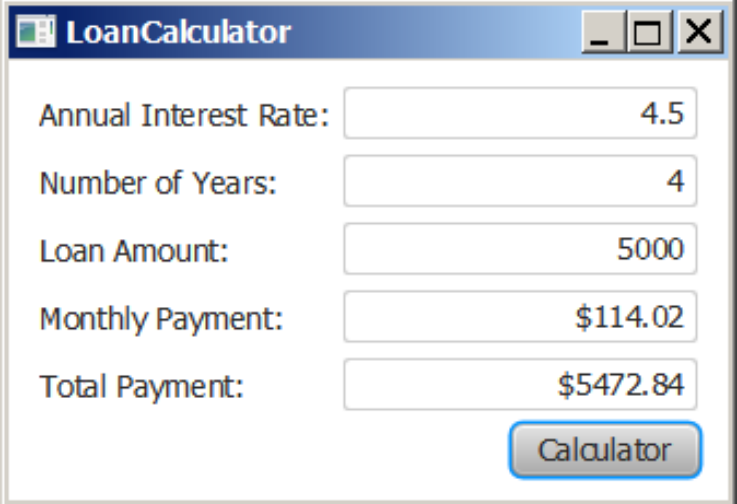
Dr. Asem Kitana

Dr. Abdallah Karakra



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.



Annual Interest Rate:	<input type="text" value="4.5"/>
Number of Years:	<input type="text" value="4"/>
Loan Amount:	<input type="text" value="5000"/>
Monthly Payment:	<input type="text" value="\$114.02"/>
Total Payment:	<input type="text" value="\$5472.84"/>

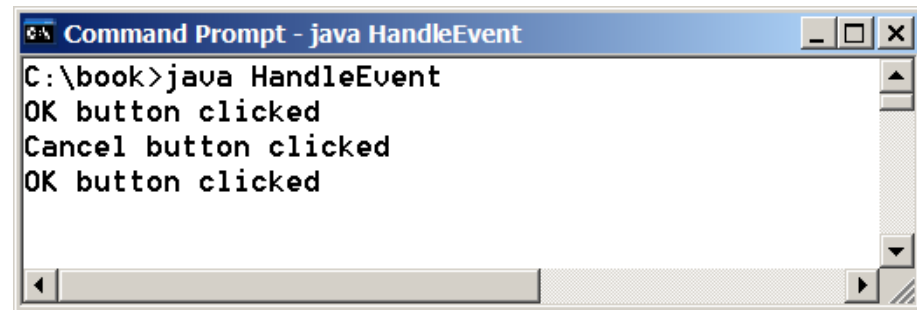
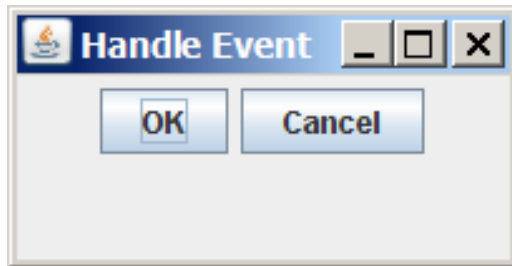
Calculator



Motivations

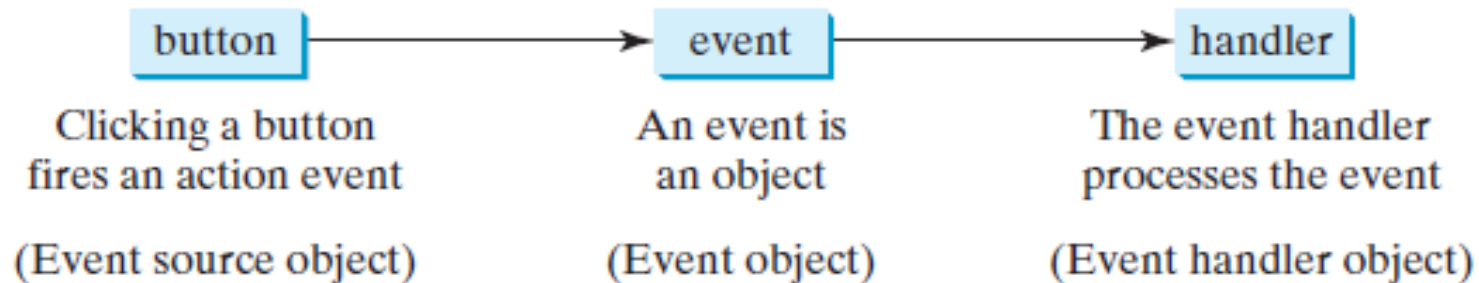
Taste of Event-Driven Programming

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

A screenshot of a Windows Command Prompt window titled "Command Prompt - java HandleEvent". The window shows the command `C:\book>java HandleEvent` being executed. The output displayed in the console is:

```
OK button clicked  
Cancel button clicked  
OK button clicked
```

Event handler



- ‡ An event handler processes the event fired from the source object.

Not all objects can be handlers for an action event. To be a handler of an action event, two requirements must be met:

1. The object 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`.
2. The `EventHandler` object `handler` must be registered with the event source object using the method `source.setAction(handler)`.



```
1 package application;
2 import javafx.application.Application;
3 import javafx.geometry.Pos;
4 import javafx.scene.Scene;
5 import javafx.scene.control.Button;
6 import javafx.scene.layout.HBox;
7 import javafx.stage.Stage;
8 import javafx.event.ActionEvent;
9 import javafx.event.EventHandler;
10
11 public class Main extends Application {
12     @Override |
13     public void start(Stage primaryStage) {
14         // Create a pane and set its properties
15         HBox pane = new HBox(10);
16         pane.setAlignment(Pos.CENTER);
```



```
17 Button btOK = new Button("OK");
18 Button btCancel = new Button("Cancel");
19 OKHandlerClass handler1 = new OKHandlerClass();
20 btOK.setOnAction(handler1);
21 CancelHandlerClass handler2 = new CancelHandlerClass();
22 btCancel.setOnAction(handler2);
23 pane.getChildren().addAll(btOK, btCancel);
24
25 // Create a scene and place it in the stage
26 Scene scene = new Scene(pane);
27 primaryStage.setTitle("HandleEvent");
28 primaryStage.setScene(scene);
29 primaryStage.show();
30 }
31 public static void main(String[] args) {
```

Create handler
Register handler



```
32     launch(args);
33 }
34 }
35
36 class OKHandlerClass implements EventHandler<ActionEvent> {
37     @Override
38     public void handle(ActionEvent e) {
39         System.out.println("OK button clicked");
40     }}
41
42 class CancelHandlerClass implements EventHandler<ActionEvent> {
43     @Override
44     public void handle(ActionEvent e) {
45         System.out.println("Cancel button clicked");
46     }}
47
```

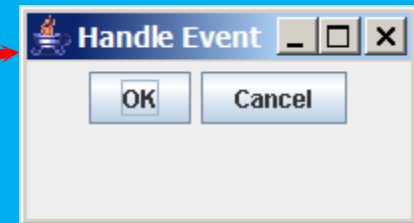
Defining
Two
handler
classes



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  
    }  
}
```

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

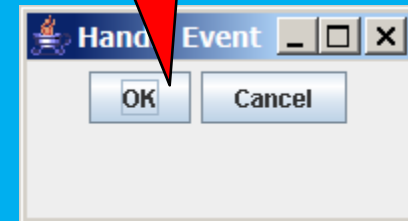


```
class OKHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```


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  
    }  
}
```

2. Click OK



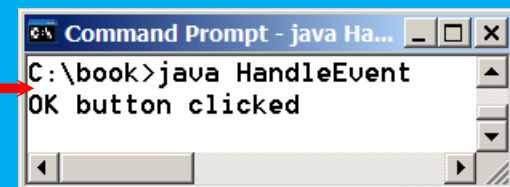
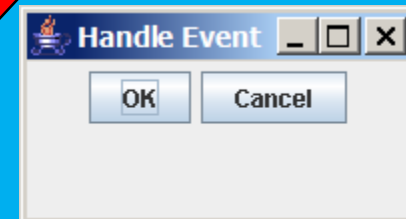
```
class OKHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

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



Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order (The program's flow execution is determined by the program's structure (and perhaps its input)).
- In event-driven programming, code is executed upon activation of events (In event-driven code, the user is responsible for determining what happens next).

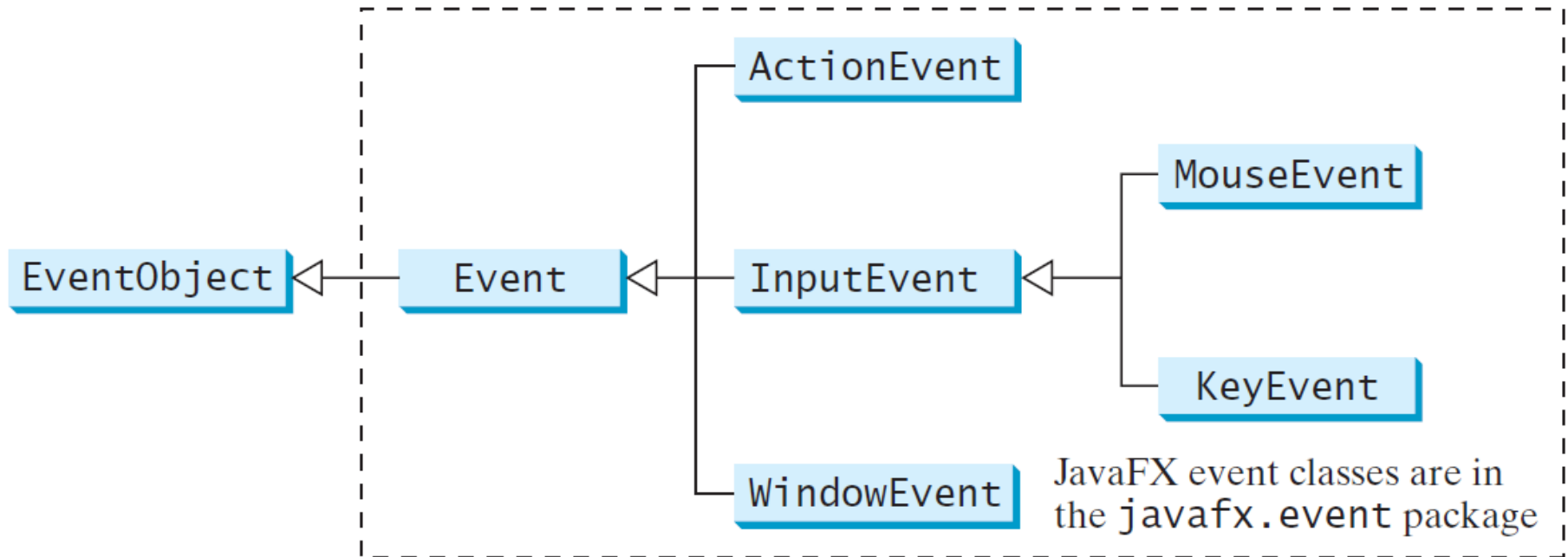


Events

- ❑ 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**.
- ❑ For example, a button is the source object for a button-clicking action event. An event is an instance of an event class. The root class of the Java event classes is `java.util.EventObject`.



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. Table 16.1 lists external user actions, source objects, and event types generated.



Selected User Actions and Handlers

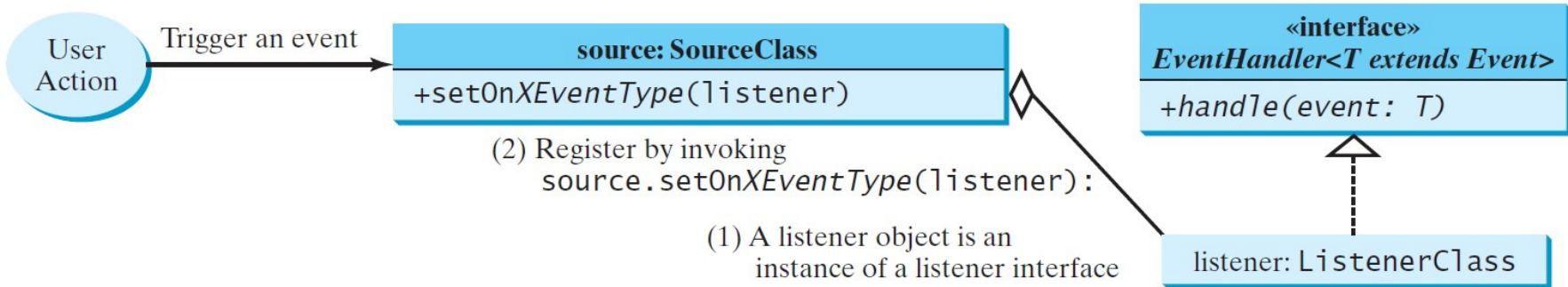
<i>User Action</i>	<i>Source Object</i>	<i>Event Type Fired</i>	<i>Event Registration Method</i>
Click a button	Button	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Press Enter in a text field	TextField	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Check or uncheck	RadioButton	ActionEvent	setOnAction(EventHandler<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>)

Selected User Actions and Handlers

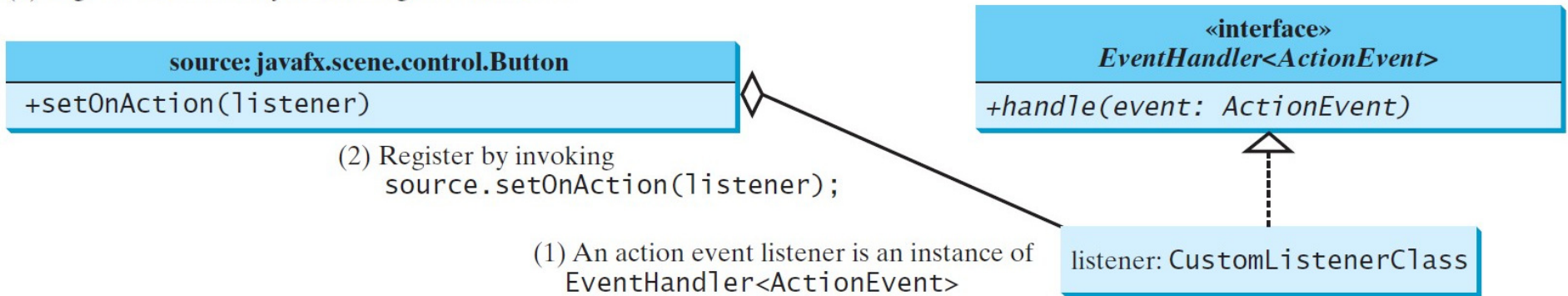
User Action	Source Object	Event Type Fired	Event Registration Method
Click a button	Button	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Press Enter in a text field	TextField	ActionEvent	setOnAction(EventHandler<ActionEvent>)
Check or uncheck	RadioButton	ActionEvent	setOnAction(EventHandler<ActionEvent>)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

What's most important for you to take away from this table is that different KINDS of source objects (GUI elements) generate different KINDS of events (e.g., buttons get clicked; not moved, like the mouse), and different kinds of events require different kinds of handlers.

The Delegation Model



(a) A generic source object with a generic event T



(b) A Button source object with an ActionEvent



The Delegation Model: Example

```
// Create the button
```

```
Button btOK = new Button("OK");
```

```
// Create handler to receive button's events
```

```
OKHandlerClass handler = new OKHandlerClass();
```

```
/*Register the handler with the button
```

```
This tells the button where to send ActionEvent*/
```

```
btOK.setAction(handler);
```



```

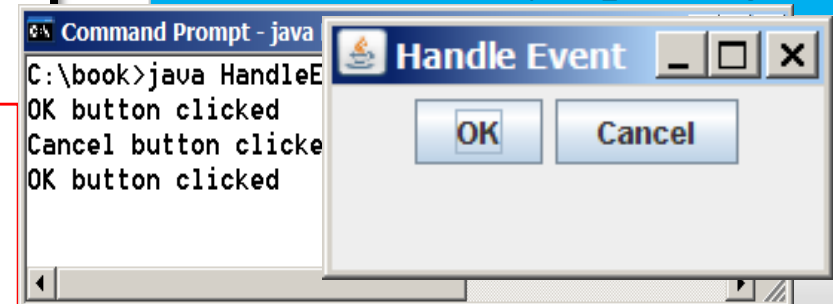
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.layout.HBox;
6  import javafx.stage.Stage;
7  import javafx.event.ActionEvent;
8  import javafx.event.EventHandler;
9
10 public class HandleEvent extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         // Create a pane and set its properties
14         HBox pane = new HBox(10);
15         pane.setAlignment(Pos.CENTER);
16         Button btOK = new Button("OK");
17         Button btCancel = new Button("Cancel");
18         OKHandlerClass handler1 = new OKHandlerClass();
19         btOK.setOnAction(handler1);
20         CancelHandlerClass handler2 = new CancelHandlerClass();
21         btCancel.setOnAction(handler2);
22         pane.getChildren().addAll(btOK, btCancel);
23
24         // Create a scene and place it in the stage
25         Scene scene = new Scene(pane);
26         primaryStage.setTitle("HandleEvent"); // Set the stage title
27         primaryStage.setScene(scene); // Place the scene in the stage
28         primaryStage.show(); // Display the stage
29     }
30 }
31
32 class OKHandlerClass implements EventHandler<ActionEvent> {
33     @Override
34     public void handle(ActionEvent e) {
35         System.out.println("OK button clicked");
36     }
37 }
38
39 class CancelHandlerClass implements EventHandler<ActionEvent> {
40     @Override
41     public void handle(ActionEvent e) {
42         System.out.println("Cancel button clicked");
43     }
44 }

```

These two handler classes (**OKHandlerClass** and **CancelHandlerClass**) are defined in the *same* **.java**

Only one class in a **.java** file can be **public**, but you can have “helper classes” in the same file as long as they’re *not* **public**.

Since these classes don’t have *any* visibility modifier, they have the default visibility - package

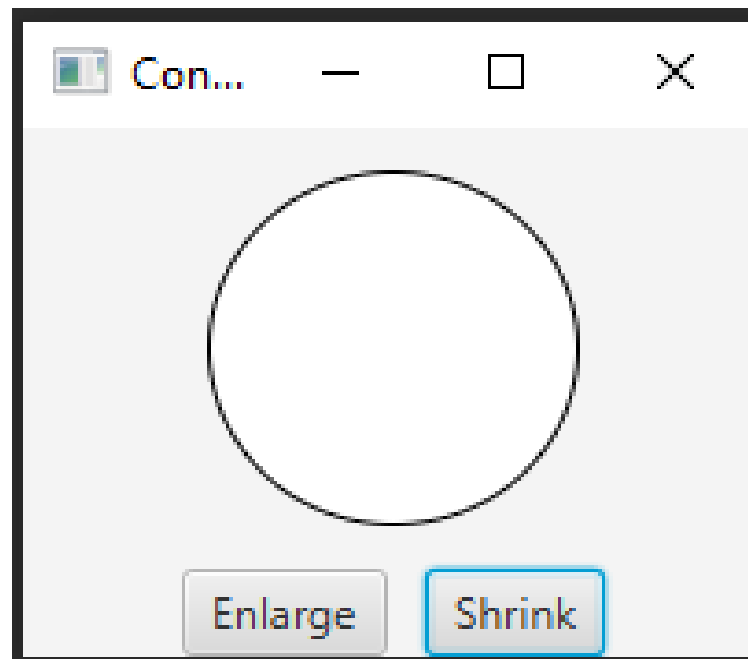


Registering Handlers and Handling Events



Example: Control Circle (without event handling)

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



Example: Control Circle (without event handling)

```
12 public class ControlCircleWithoutEventHandling extends Application {
13     @Override // Override the start method in the Application class
14     public void start(Stage primaryStage) {
15         StackPane pane = new StackPane();
16         Circle circle = new Circle(50);
17         circle.setStroke(Color.BLACK);
18         circle.setFill(Color.WHITE);
19         pane.getChildren().add(circle);
20
21         HBox hBox = new HBox();
22         hBox.setSpacing(10);
23         hBox.setAlignment(Pos.CENTER);
24         Button btEnlarge = new Button("Enlarge");
25         Button btShrink = new Button("Shrink");
26         hBox.getChildren().add(btEnlarge);
27         hBox.getChildren().add(btShrink);
28
29         BorderPane borderPane = new BorderPane();
30         borderPane.setCenter(pane);
31         borderPane.setBottom(hBox);
32         BorderPane.setAlignment(hBox, Pos.CENTER);
33
34         // Create a scene and place it in the stage
35         Scene scene = new Scene(borderPane, 200, 150);
36         primaryStage.setTitle("ControlCircle"); // Set
37         primaryStage.setScene(scene); // Place the scene
38         primaryStage.show(); // Display the stage
39     }
49 }
```

Our program starts by extending **Application**, and overriding the **start** method, where we build the UI

The **Circle** goes into a **StackPane**

The two **Buttons** get created and put into an **HBox** pane

The **StackPane** with the **Circle** goes into the *middle* section of a **BorderPane**, and the **HBox** with the two **Buttons** goes into the *bottom* section.

Finally, add the **BorderPane** to the **Scene**, and the **Scene** to the **Stage**, and make the **Stage** visible

That gets the UI created, but nothing is configured to handle events (yet)

```
1 package application;
2 import javafx.application.Application;
3 import javafx.geometry.Pos;
4 import javafx.scene.Scene;
5 import javafx.scene.control.Button;
6 import javafx.scene.layout.StackPane;
7 import javafx.scene.layout.HBox;
8 import javafx.scene.layout.BorderPane;
9 import javafx.scene.paint.Color;
10 import javafx.scene.shape.Circle;
11 import javafx.stage.Stage;
12
13 public class Main extends Application {
14     @Override
15     public void start(Stage primaryStage) {
16         StackPane pane = new StackPane();
17         Circle circle = new Circle(50);
18         circle.setStroke(Color.BLACK);
19         circle.setFill(Color.WHITE);
```

```
20 pane.getChildren().add(circle);
21
22 HBox hBox = new HBox();
23 hBox.setSpacing(10);
24 hBox.setAlignment(Pos.CENTER);
25 Button btEnlarge = new Button("Enlarge");
26 Button btShrink = new Button("Shrink");
27 hBox.getChildren().add(btEnlarge);
28 hBox.getChildren().add(btShrink);
29
30 BorderPane borderPane = new BorderPane();
31 borderPane.setCenter(pane);
32 borderPane.setBottom(hBox);
33 BorderPane.setAlignment(hBox, Pos.CENTER);
34
```

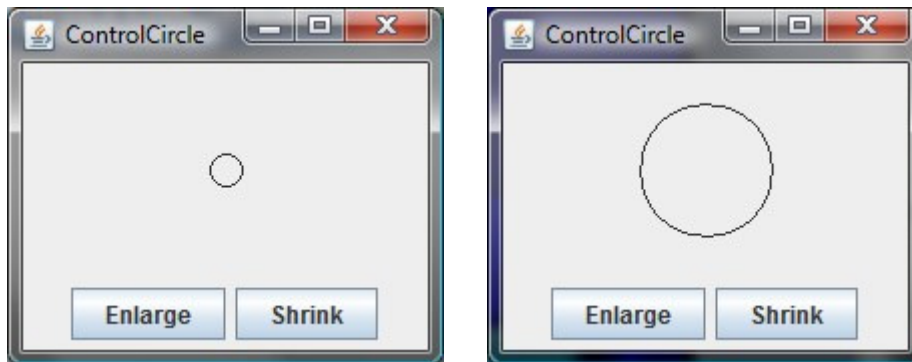


```
35 // Create a scene and place it in the stage
36 Scene scene = new Scene(borderPane, 200, 150);
37 primaryStage.setTitle("ControlCircle"); // Set the stage title
38 primaryStage.setScene(scene); // Place the scene in the stage
39 primaryStage.show(); // Display the stage
40 }
41 public static void main(String[] args) {
42     launch(args);
43 }
44 }
45
```



Example: Control Circle (with event handling)

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



```

14 public class ControlCircle extends Application {
15     private CirclePane circlePane = new CirclePane();
16
17     @Override // Override the start method in the Application class
18     public void start(Stage primaryStage) {
19         // Hold two buttons in an HBox
20         HBox hBox = new HBox();
21         hBox.setSpacing(10);
22         hBox.setAlignment(Pos.CENTER);
23         Button btEnlarge = new Button("Enlarge");
24         Button btShrink = new Button("Shrink");
25         hBox.getChildren().add(btEnlarge);
26         hBox.getChildren().add(btShrink);
27
28         // Create and register the handler
29         btEnlarge.setOnAction(new EnlargeHandler());
30
31         BorderPane borderPane = new BorderPane();
32         borderPane.setCenter(circlePane);
33         borderPane.setBottom(hBox);
34         BorderPane.setAlignment(hBox, Pos.CENTER);
35
36         // Create a scene and place it in the stage
37         Scene scene = new Scene(borderPane, 200, 150);
38         primaryStage.setTitle("ControlCircle"); // Set the stage title
39         primaryStage.setScene(scene); // Place the scene in the stage
40         primaryStage.show(); // Display the stage
41     }
42
43     class EnlargeHandler implements EventHandler<ActionEvent> {
44         @Override // Override the handle method
45         public void handle(ActionEvent e) {
46             circlePane.enlarge();
47         }
48     }
49 }
50
51 class CirclePane extends StackPane {
52     private Circle circle = new Circle(50);
53
54     public CirclePane() {
55         getChildren().add(circle);
56         circle.setStroke(Color.BLACK);
57         circle.setFill(Color.WHITE);
58     }
59
60     public void enlarge() {
61         circle.setRadius(circle.getRadius() + 2);
62     }
63
64     public void shrink() {
65         circle.setRadius(circle.getRadius() > 2 ?
66             circle.getRadius() - 2 : circle.getRadius());
67     }

```

The new **CirclePane** class will be an extension of a **StackPane** that contains the circle

Again, because we are not defining a second **public** class, this package-level-visibility class can reside in the same **.java** sourcefile.

Because it's an extension of the **StackPane**, the **CirclePane** can access its own list of children, and add a circle in its constructor

The **CirclePane's** **enlarge** and **shrink** methods simply add or subtract 2 to the circle's current radius. In the **shrink** method, it checks before subtracting 2 to make sure it's over 2 (the radius can't go negative, and it's probably a good idea to keep it from going to zero, so **> 2**, as opposed to **>= 2**, is a wise idea.

```
1 package application;
2
3 import javafx.application.Application;
4 import javafx.event.ActionEvent;
5 import javafx.event.EventHandler;
6 import javafx.geometry.Pos;
7 import javafx.scene.Scene;
8 import javafx.scene.control.Button;
9 import javafx.scene.layout.StackPane;
10 import javafx.scene.layout.HBox;
11 import javafx.scene.layout.BorderPane;
12 import javafx.scene.paint.Color;
13 import javafx.scene.shape.Circle;
14 import javafx.stage.Stage;
15
16 public class Main extends Application {
17     private CirclePane circlePane = new CirclePane();
18
```

```
18
19 @Override //
20 public void start(Stage primaryStage) {
21     // Hold two buttons in an HBox
22     HBox hBox = new HBox();
23     hBox.setSpacing(10);
24     hBox.setAlignment(Pos.CENTER);
25     Button btEnlarge = new Button("Enlarge");
26     Button btShrink = new Button("Shrink");
27     hBox.getChildren().add(btEnlarge);
28     hBox.getChildren().add(btShrink);
29
30     // Create and register the handler
31     btEnlarge.setOnAction(new EnlargeHandler());
32
33     btShrink.setOnAction(new ShrinkHandler());
34
35     BorderPane borderPane = new BorderPane();
36     borderPane.setCenter(circlePane);
```

```
37 borderPane.setBottom(hBox);
38 BorderPane.setAlignment(hBox, Pos.CENTER);
39
40 // Create a scene and place it in the stage
41 Scene scene = new Scene(borderPane, 200, 150);
42 primaryStage.setTitle("ControlCircle"); // Set the stage title
43 primaryStage.setScene(scene); // Place the scene in the stage
44 primaryStage.show(); // Display the stage
45 }
46 public static void main(String[] args) {
47     launch(args);
48 }
49
50 class EnlargeHandler implements EventHandler<ActionEvent> {
51     @Override // Override the handle method
52     public void handle(ActionEvent e) {
53         circlePane.enlarge();
54     }
55 }
```

```
57● class ShrinkHandler implements EventHandler<ActionEvent> {
58● @Override // Override the handle method
▲59 public void handle(ActionEvent e) {
60     circlePane.shrink();
61 }
62 class CirclePane extends StackPane {
63 private Circle circle = new Circle(50);
64● public CirclePane() {
65     getChildren().add(circle);
66     circle.setStroke(Color.BLACK);
67     circle.setFill(Color.WHITE);
68 }
69● public void enlarge() {
70     circle.setRadius(circle.getRadius() + 2);
71 }
72● public void shrink() {
73     circle.setRadius(circle.getRadius() > 2 ?
74     circle.getRadius() - 2 : circle.getRadius());
75 }
```

Inner Classes

