Event-Driven Programming, Inner Classes, and Lambda Expressions

Dr. Abdallah Karakra |Comp 2311 | Masri504 6/16/2023

BIRZEIT UNIVERSITY

Index

Chapter 15

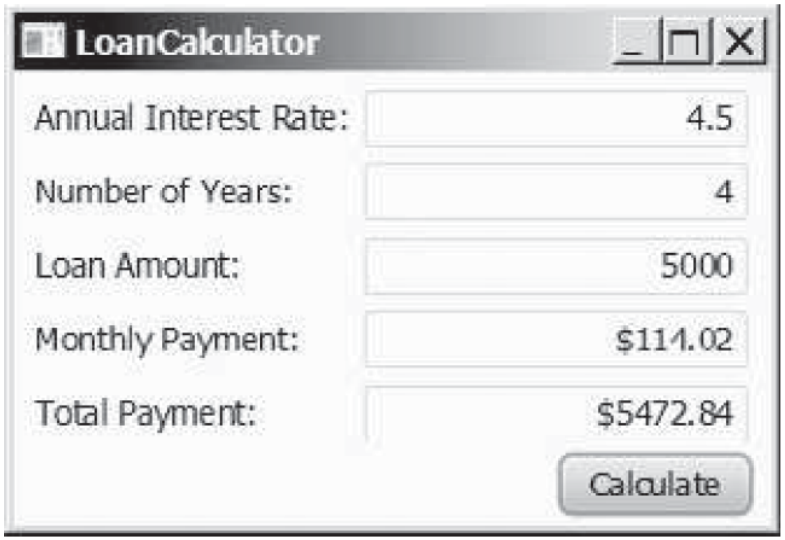
- All Sections (15.1 – 14.13)

CHAPTER 15

Event-Driven Programming and Animations

Introduction to Event-Driven Software

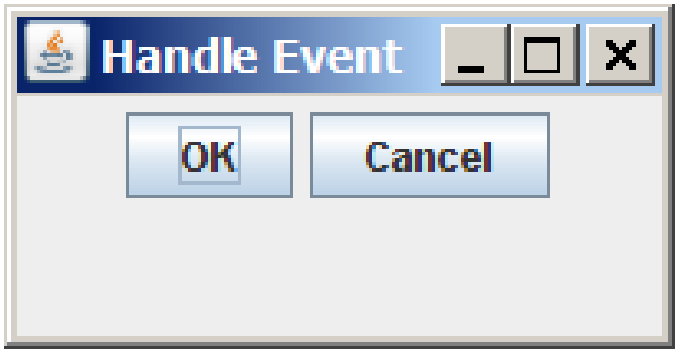
Introduction



- Suppose you want to develop a GUI application to calculate loan payments:
- The user should be able type the pertinent values into the corresponding boxes, and then click "calculate"
- How can our program tell when the "Calculate" button has been pressed (clicked on), so we can run the code to do the calculation?

Introduction

- Let's run through a simple example to get a taste of what's involved. We'll create a stage with two buttons – "OK" and "Cancel"
- When the "OK" button is clicked, we'll output "OK Button Clicked" on the console
- When the "Cancel" button is clicked, we'll output "Cancel button clicked" on the console



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)).

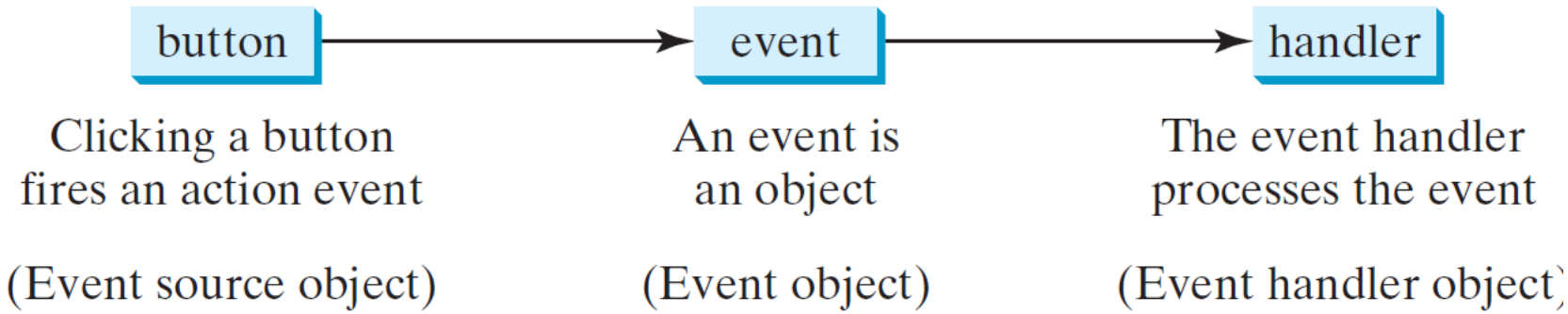
In **event-driven programming**, code is executed upon activation of events(**In *event-driven code*, the *user* is responsible for determining what happens next**).

Handling GUI Events

Source object (e.g., button, polygon, image, etc.)

Event object (e.g., mouse click, mouse pointer over object, type characters into a textfield, etc)

Listener object contains a method for processing the event



Handling GUI Events

- **Not all objects can be handlers** for some action event. To be a handler of an event, **there are two requirements**:

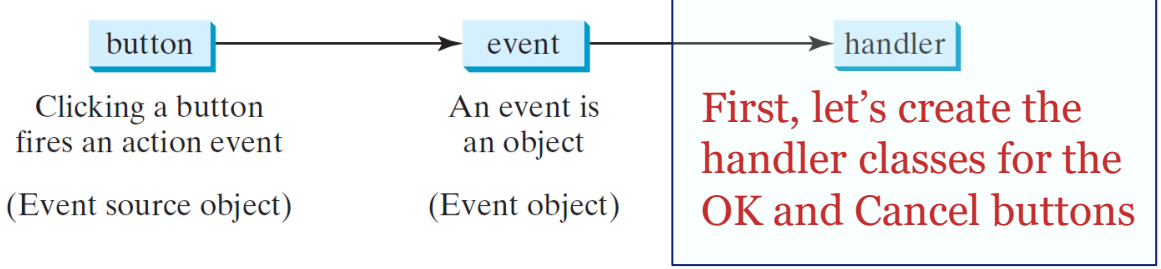
  1. The object must implement the interface EventHandler<T extends Event>, which defines the common behavior for all action handlers
  2. The EventHandler object handler must be registered with the event source object using the source.setOnAction(handler) method

- The EventHandler<ActionEvent> interface contains the method handle(ActionEvent) for processing the event -- your handler class must override this method to respond to the event

Example: Event-Driven Programming

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

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

- Creating the OK handler class by implementing the EventHandler interface
- Overriding the EventHandler's handle method
- Handling the event is simply printing the statement "OK button clicked" to the console
- Creating the Cancel handler class by implementing the EventHandler interface
- Overriding the EventHandler's handle method
- Handling the event is simply printing the statement "Cancel button clicked" to the console

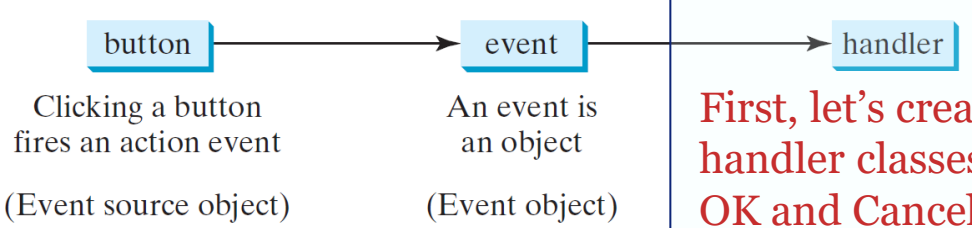


Example: Event-Driven Programming

```
class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  pu

  }
}

clas
  @O
  pu

  }
}
```

Now that we have created the handler classes for each button, let's now use those classes in the Main

- Creating the OK handler class by implementing the EventHandler interface
- Overriding the EventHandler's handle method
- event is ng the K button e console
- Cancel by g the r interface
- Overriding the EventHandler's handle method
- Handling the event is simply printing the statement "Cancel button clicked" to the console



First, let's create the handler classes for the OK and Cancel buttons

Example: Event-Driven Programming

```java
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public class HandleEvent extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    HBox pane = new HBox(10);
    pane.setAlignment(Pos.CENTER);
    Button btOK = new Button("OK");
    Button btCancel = new Button("Cancel");
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    pane.getChildren().addAll(btOK, btCancel);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("HandleEvent"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}

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

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

Import the following for the events and handlers

Creating and setting up pane
Creating the OK button
Creating the Cancel button

Creating the OK button's handler from the class created (recall the listener must be an instance of a listener interface)

Registering the newly created handler with the OK button *(recall a listener must be registered with a source object)*

Creating the Cancel button's handler from the class created

Registering the newly created handler with the Cancel button

Adding the OK and Cancel buttons to the pane

Create the Scene, place it in the stage.....

Example: Event-Driven Programming

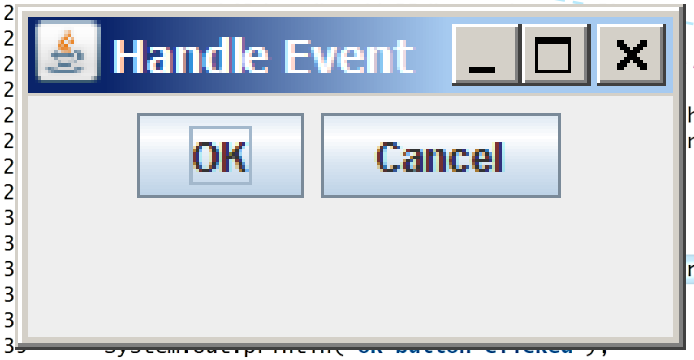
Import the following for the events and handlers

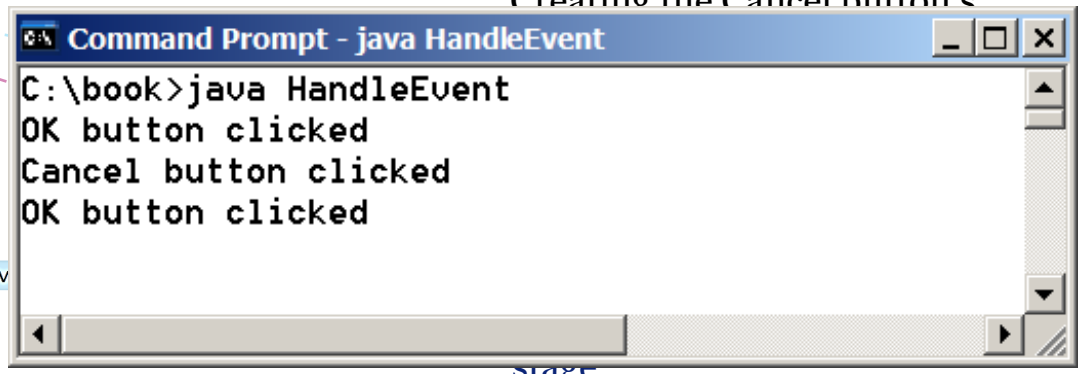
Creating and setting up pane
Creating the OK button
Creating the Cancel button

Creating the OK button's handler from the class created (recall the

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public clas
  @Override
  public vo
    // Crea
    HBox pa
    pane.se
    Button
    Button
    OKHandl
    btOK.se
    CancelH
    btCancel.setOnAction(handler2);
```

Now the OK and Cancel buttons can "listen" for mouse click events as the program runs – and react accordingly.






```
    }
  }

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

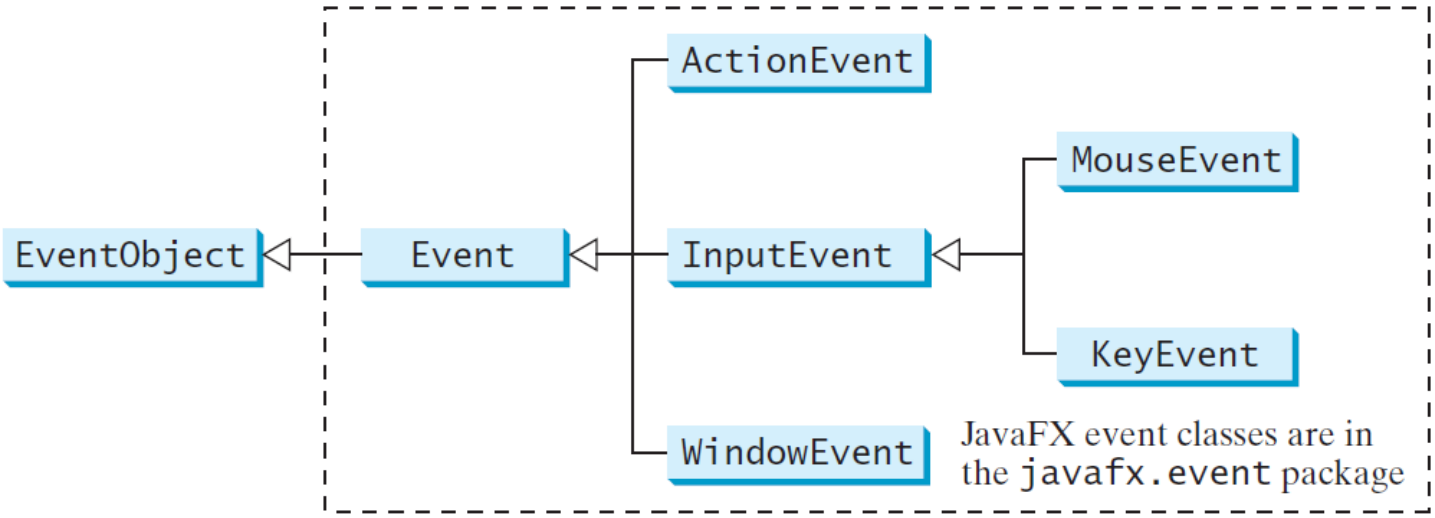
Events and Event Sources

Events and Event Sources

- In event-driven programming, events drive (determine) the program's execution
- An event is a signal (message) that something has happened
- Some events (like button clicks, key presses, mouse movements) are triggered by user action
- Some events can be generated by internal program activities

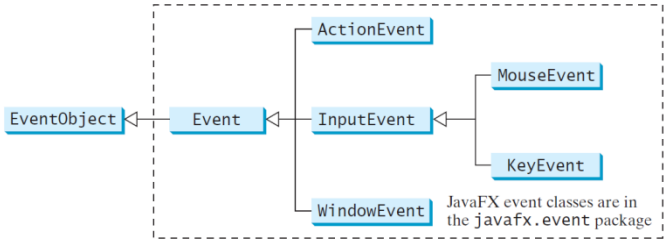
Events and Event Sources

- `EventObject` hierarchy:



- `EventObject` has a `getSource()` method (so do its descendants), so an event handler can tell who generated an event it receives (caller ID)

Selected User Actions and Handlers



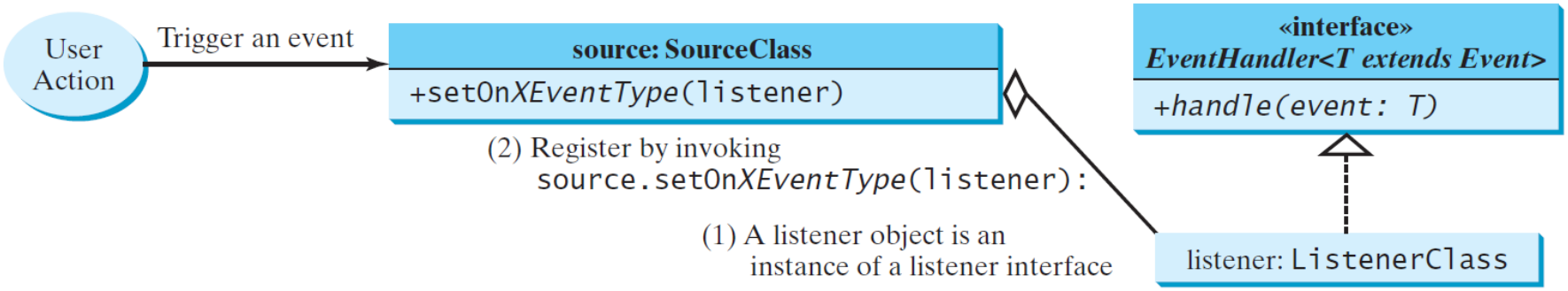
User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler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

Selected User Actions and Handlers

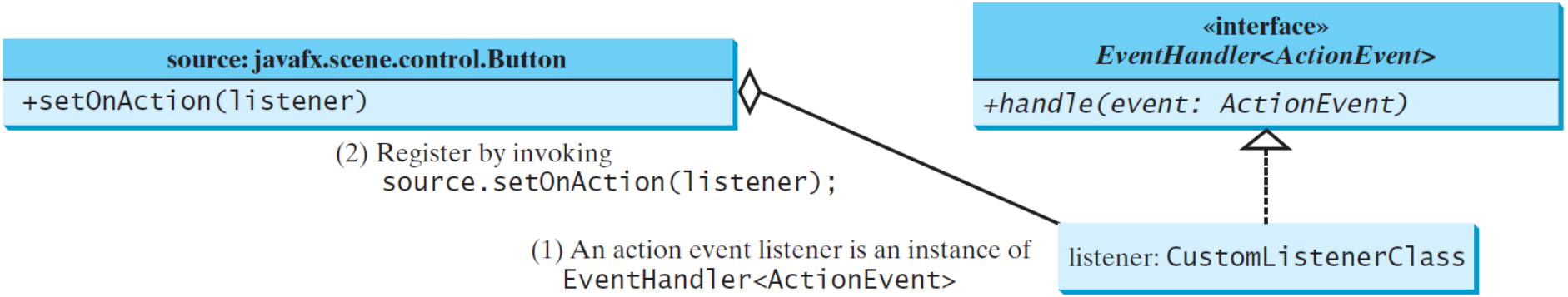
User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select			
Mouse			nt>)
Mouse			ent>)
Mouse			nt>)
Mouse			nt>)
Mouse			t>)
Mouse			>)
Mouse			nt>)
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`

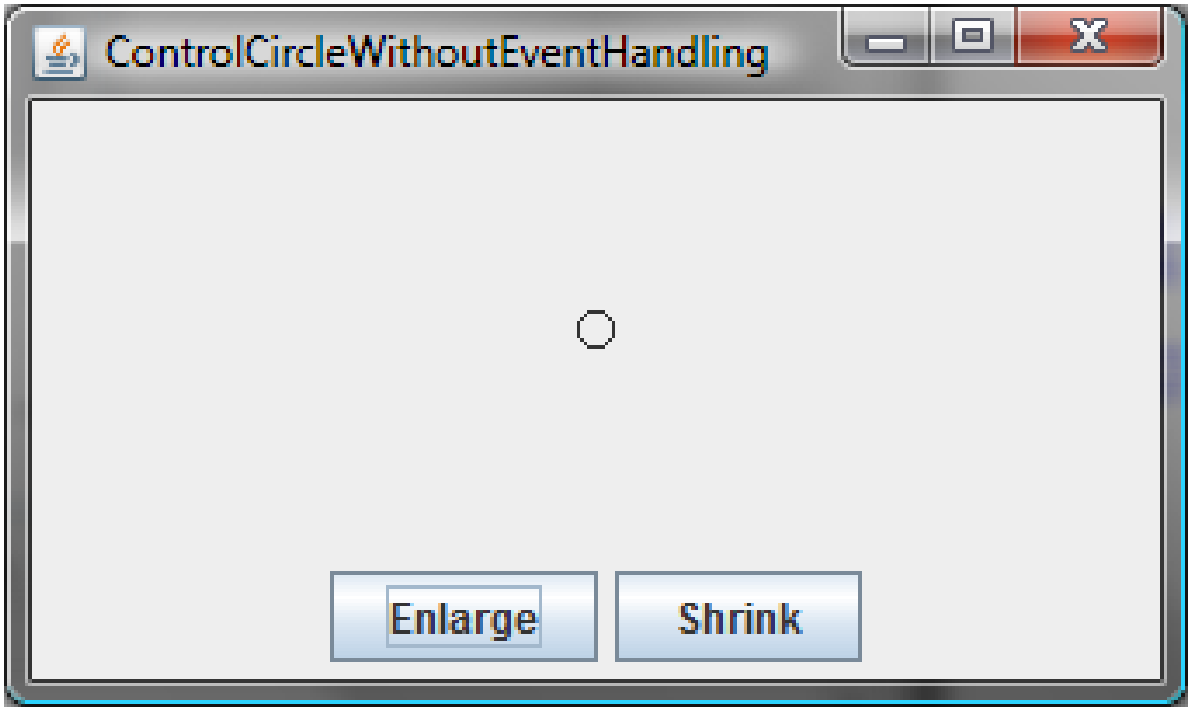
Registering Handlers and Handling Events

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.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.



Example: First Version for ControlCircle (no listeners)

```
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 the stage title
37     primaryStage.setScene(scene); // Place the scene in the stage
38     primaryStage.show(); // Display the stage
39   }
49 }
```

Example: First Version for ControlCircle (no listeners)

```java
public class ControlCircleWithoutEventHandling extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    Circle circle = new Circle(50);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
    pane.getChildren().add(circle);

    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

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

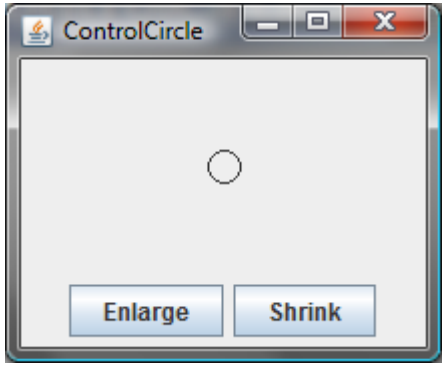
The `StackPane` with the `Circle` goes into the *middle* section of a `BorderPane`, and the `HBox` with the two `Buttons` goes into the *bottom* section.

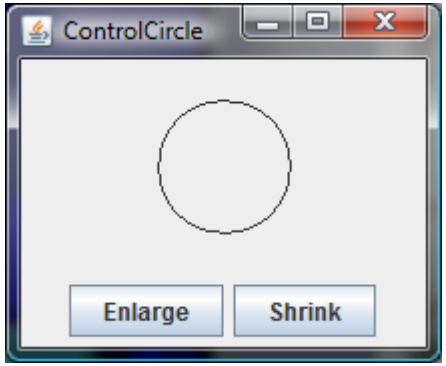
Finally, add the `BorderPane` to the `Scene`, and the `Scene` to the `Stage`, and make the `Stage` visible

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

Example: Second Version for Control Circle (with listener for Enlarge)

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





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

```java
public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(new EnlargeHandler());

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 700, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override // Override the handle method
    public void handle(ActionEvent e) {
      circlePane.enlarge();
    }
  }
}

class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
    getChildren().add(circle);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
  }

  public void enlarge() {
    circle.setRadius(circle.getRadius() + 2);
  }

  public void shrink() {
    circle.setRadius(circle.getRadius() > 2 ?
      circle.getRadius() - 2 : circle.getRadius());
  }
}
```

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` source file.

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.

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

```java
public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(new EnlargeHandler());

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override // Override the handle method
    public void handle(ActionEvent e) {
      circlePane.enlarge();
    }
  }
}

class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
    getChildren().add(circle);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
  }

  public void enlarge() {
    circle.setRadius(circle.getRadius() + 2);
  }

  public void shrink() {
    circle.setRadius(circle.getRadius() > 2 ?
      circle.getRadius() - 2 : circle.getRadius());
  }
}
```

Armed with our new, fully-functional `CirclePane` class, we can just make it a field of the `ControlCircle` class. We instantiate it as we declare it

When we build the `BorderPane`, we add the `CirclePane` to its center section, just a before

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

```
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   }
68 }
```

Now things get a bit more interesting.

We know that the `Button`s will need a class they can fire their events to.

We create the `EnlargeHandler` class to handle the `ActionEvent`s that will come from the "Enlarge" `Button`.

The `EnlargeHandler` class has a `handle` method, which will only have to call the `CirclePane`'s enlarge method.

Look closely, and you'll realize that the `ControlCircle` class runs from line 14 to 49

The `EnlargeHandler` class runs from line 43 through line 48 – it's INSIDE the `ControlCircle` class!

WHAT !?!

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

```java
public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(new EnlargeHandler());

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override // Override the handle method
    public void handle(ActionEvent e) {
      circlePane.enlarge();
    }
  }
}

class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
    getChildren().add(circle);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
  }

  public void enlarge() {
    circle.setRadius(circle.getRadius() + 2);
  }

  public void shrink() {
    circle.setRadius(circle.getRadius() > 2 ?
      circle.getRadius() - 2 : circle.getRadius());
  }
}
```

So far, classes have contained fields and methods, period.

In this example, we have a class that also contains another class. This nesting of classes is called creating an *inner class*. We'll talk more about this in the next section.

One of the reasons to have the inner class is related to variable scope. Because `circlePane` is a field, its scope is class-wide, so inside the `EnlargeHandler`'s handle method, `circlePane` is in-scope, so we can *see* it to call its `enlarge` method

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

```
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   }
68 }
```

So, now that we have a handler class for our "enlarge" `Button`, we need to instantiate it and tell the button to send its `ActionEvents` to it.

Line 29 does both in one shot

This code, as-written, doesn't have any provision for the `shrink` button.

The book leaves that as an exercise, but by now, it should be obvious what we need to do:

Create a second inner class, `ShrinkHandler`, that works just like `EnlargeHandler`, except that its `handle` method should call the `circlePane`'s `shrink` method, rather than its `enlarge` method.

The other thing we need to do is to instantiate the `ShrinkHandler` class, and register it with the "shrink" `Button`.

This is just a copy of line 29, except that it's `btnShrink` and new `ShrinkHandler()`

Inner Classes

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

```
public class Test
{
   …
}
```

Test.java → Test.class

```
public class A
{
   …
}
```

A.java → A.class

```
public class Test
{
   …
   // inner class
   class A
   {
      …
   }
}
```

Test.java → Test.class
and → Test$A.class

Inner Classes, cont.

```java
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

```java
public class Test {
  ...

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

(b)

```java
// 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.)

- Inner classes can make programs simple and concise.
- An inner class supports the work of its containing outer class and is compiled into a class named

***OuterClassName*$*InnerClassName*.class**.

For example, the inner class InnerClass in OuterClass is compiled into *OuterClass*$*InnerClass*.class .

→ `Test$A.class`

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 Class Handlers

Anonymous Inner Classes

- "An anonymous inner class is an inner class without a name. It combines **defining an inner class and creating an instance of the class into one step**"
- anonymous inner class combines **a class's definition and instantiation in one step**

Anonymous Inner Classes

```
public void start(Stage primaryStage)
{
   // Several Lines Omitted
   btnEnlarge.setOnAction(new EnlargeHandlner());
}


// EnlargeHandler as an inner class
class EnlargeHandler implements EventHandler<ActionEvent>
{
   @Override
   public void handle(ActionEvent e)
   {
      circlePane.enlarge();
   }
}
```

Anonymous Inner Classes

```
public void start(Stage primaryStage)
{
   // Several Lines Omitted
   btnEnlarge.setOnAction(new EnlargeHandlner());
}


// EnlargeHandler as an inner class
class EnlargeHandler implements EventHandler<ActionEvent>
{
   @Override
   public void handle(ActionEvent e)
   {
      circlePane.enlarge();
   }
}
```

Anonymous Inner Classes

```
public void start(Stage primaryStage)
{
   // Several Lines Omitted
   btnEnlarge.setOnAction(new EnlargeHandlner());
}


// EnlargeHandler as an inner class
class EnlargeHandler implements EventHandler<ActionEvent>
{
   @Override
   public void handle(ActionEvent e)
   {
      circlePane.enlarge();
   }
}
```

Anonymous Inner Classes

```
public void start(Stage primaryStage)
{
   // Several Lines Omitted
   btnEnlarge.setOnAction(new EnlargeHandlner());
}


// EnlargeHandler as an inner class
class EnlargeHandler implements EventHandler<ActionEvent>
{
   @Override
   public void handle(ActionEvent e)
   {
      circlePane.enlarge();
   }
}
```

Anonymous Inner Classes

```
public void start(Stage primaryStage)
{
   // Several Lines Omitted
   btnEnlarge.setOnAction(new EventHandler<ActionEvent>
                          {
                              @Override
                              public void handle(ActionEvent e)
                              {
                                  circlePane.enlarge();
                              }
                           });
}
```

Anonymous Inner Classes

- Since an **anonymous inner class is a special kind of inner class, it is treated like an inner class** with the following four features:

1. An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit `extends` or `implements` clause.

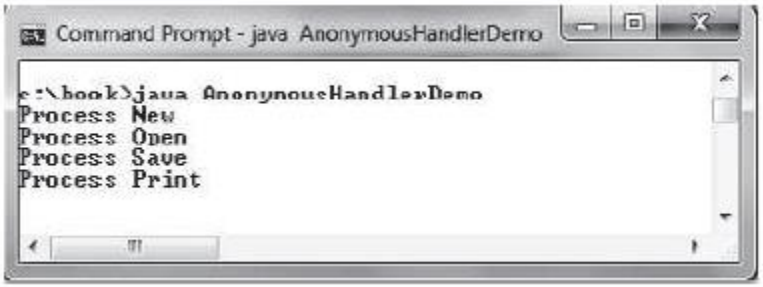
2. An anonymous inner class must implement all the abstract methods in the superclass or in the interface (it must "go concrete").

3. 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()`.

4. An anonymous inner class is compiled into a class named `OuterClassName$n.class`. For example, if the outer class `Test` has two anonymous inner classes, they are compiled into `Test$1.class` and `Test$2.class`.

Example:Anonymous Inner Classes

- When each button is clicked, it produces output on the console (although we can certainly have its click event handler do anything we want):



Example:Anonymous Inner Classes

```
10 public class AnonymousHandlerDemo extends Application {
11   @Override // Override the start method in the Application class
12   public void start(Stage primaryStage) {
13     // Hold two buttons in an HBox
14     HBox hBox = new HBox();
15     hBox.setSpacing(10);
16     hBox.setAlignment(Pos.CENTER);
17     Button btNew = new Button("New");
18     Button btOpen = new Button("Open");
19     Button btSave = new Button("Save");
20     Button btPrint = new Button("Print");
21     hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);
22
23     // Create and register the handler
24     btNew.setOnAction(new EventHandler<ActionEvent>() {
25       @Override // Override the handle method
26       public void handle(ActionEvent e) {
27         System.out.println("Process New");
28       }
29     });
30
31     btOpen.setOnAction(new EventHandler<ActionEvent>() {
32       @Override // Override the handle method
33       public void handle(ActionEvent e) {
34         System.out.println("Process Open");
35       }
36     });
37
38     btSave.setOnAction(new EventHandler<ActionEvent>() {
39       @Override // Override the handle method
40       public void handle(ActionEvent e) {
41         System.out.println("Process Save");
42       }
43     });
44
45     btPrint.setOnAction(new EventHandler<ActionEvent>() {
46       @Override // Override the handle method
47       public void handle(ActionEvent e) {
48         System.out.println("Process Print");
49       }
50     });
51
52     // Create a scene and place it in the stage
53     Scene scene = new Scene(hBox, 300, 50);
54     primaryStage.setTitle("AnonymousHandlerDemo"); // Set title
55     primaryStage.setScene(scene); // Place the scene in the stage
56     primaryStage.show(); // Display the stage
57   }
58 }
```

I've omitted the `import` statements from the top

Lines 14 – 21 create an `HBox` and four `Button`s, which it adds to the `HBox`.

The "New" button (`btnNew`) needs to be registered with the object that will receive (and handle) its `ActionEvents`.

That object will have to be an instance of a class that `inplements EventHanlder<ActionEvent>`

That's precisely what the blue-shaded code in lines 24 – 29 IS

The same format is repeated to register an anonymous inner `EventHandler<ActionEvent>` class for `btnOpen`, `btnSave`, and `btnPrint`

By "embedding" the handler classes inside the class with the UI, we eliminate the need for several explicitly-defined classes that will be used only to handle events for one object!

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

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
public class ControlCircle extends Application {
    @Override // Override the start method in the Application class
      public void start(Stage primaryStage) {
        StackPane pane = new StackPane();
        Circle circle = new Circle(50);
        circle.setStroke(Color.BLACK);
        circle.setFill(Color.WHITE);
        pane.getChildren().add(circle);

        HBox hBox = new HBox();
        hBox.setSpacing(10);
        hBox.setAlignment(Pos.CENTER);
        Button btEnlarge = new Button("Enlarge");
        btEnlarge.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                circle.setRadius(circle.getRadius()+2);
            }
        });
```

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

```
    Button btShrink = new Button("Shrink");
    btShrink.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event){
            circle.setRadius(circle.getRadius()-2);
        }
    });

    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

Using Lambda Expressions to Simplify Event Handling

Lambda Expressions and Events

- Lambda Expressions (new to Java 8) can be considered **an anonymous inner class with an abbreviated syntax**.
- The compiler treats lambda expressions like an **object created from an anonymous inner class**

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

(a) Anonymous inner class event handler

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

(b) Lambda expression event handler

Lambda Expressions and Events

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

(a) Anonymous inner class event handler

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

(b) Lambda expression event handler

- The compiler knows that the parameter of the `setOnAction` method must be something of type `EventHandler<ActionEvent>`, and that particular interface has only one abstract method, so the code between the braces **must** be the statements for that method
- Let's revisit the anonymous inner class handler:

Lambda Expressions and Events

```java
// Create and register the handler
btNew.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process New");
  }
});

btOpen.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process Open");
  }
});

btSave.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process Save");
  }
});

btPrint.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process Print");
  }
});
```

In the original code, each button had its own complete anonymous inner class to handle the `ActionEvent` from its corresponding button

```java
// Create and register the handler
btNew.setOnAction((ActionEvent e) -> {
  System.out.println("Process New");
});

btOpen.setOnAction((e) -> {
  System.out.println("Process Open");
});

btSave.setOnAction(e -> {
  System.out.println("Process Save");
});

btPrint.setOnAction(e -> System.out.println("Process Print"));
```

In the updated version, we show four different ways (styles) of expressing the registration of the event handler using lambda expressions

The first explicitly gives the type of `e`

The second omits the type, because the compiler can infer that `e` is of type `ActionEvent`

The third omits the parentheses, because there is only one parameter

The fourth omits the braces, because there is only one line of code (much like a one-line then or else clause of an `if` / `then` / `else`, or a one-statement loop body)

Lambda expressions make for cleaner code!

Case Study: The Loan Calculator

Mouse Events

Mouse Events

- Mouse events are fired whenever the mouse…
  - …button goes down (`MousePressed`)
  - …button comes back up (`MouseReleased`)
  - …button is clicked (a down/up cycle – `MouseClicked`)
  - …first enters an element (`MouseEntered`)
  - …leaves an element (`MouseExited`)
  - …moves while over an element (`MouseMoved`)
  - …is dragged (moved with the mouse button held down – `MouseDragged`)

The `MouseEvent` Class

MouseEvent object captures the event, such as **the number of clicks** associated with it**, the location (the x- and y-coordinates)** of the mouse, or **which mouse button was pressed**)

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.

Mouse Events

- JavaFX (currently) supports 3 buttons, left, middle, and right (some mice have more)

- We can tell *which* mouse button was pressed by using the

`MouseEvent`'s `getButton()` method and comparing the result to:

```
MouseButton.PRIMARY,   //left button
MouseButton.SECONDARY,  // right button
MouseButton.MIDDLE, or // middle button
MouseButton.NONE  // none
```

- `NONE` is on the list simply because some mouse events (like `MouseMoved`) don't involve a button

- Example: getButton() == MouseButton.SECONDARY tests if the right button was pressed. You can also use the isPrimaryButtonDown(), isSecondaryButtonDown(), and isMiddleButtonDown() to test if the primary button, second button, or middle button is pressed

Mouse Events

- The book presents a sample program that lets the user drag (click, holding the primary (left?) mouse button down, and moving the mouse before releasing the button) a `Text` node around on the pane:

- The (surprisingly short) code is on the next slide

Example: MouseEventDemo

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MouseEventDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    Pane pane = new Pane();
    Text text = new Text(20, 20, "Programming is fun");
    pane.getChildren().addAll(text);
    text.setOnMouseDragged(e -> {
      text.setX(e.getX());
      text.setY(e.getY());
    });

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 300, 100);
    primaryStage.setTitle("MouseEventDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

When the mouse is dragging the text, set the text's X/Y location to the X/Y location of the mouse (which we get from the event, e)

Key Events

Key Events

- The user can also interact with the GUI through the keyboard.
- We can capture (handle) events that are generated by the keyboard on a key-by-key basis, without having to wait for the user to press `Enter`, as we do with the `Scanner`
- We can tell when keys go down (`KEY_PRESSED`), come back up (`KEY_RELEASED`), or make a down-and-up round-trip (`KEY_TYPED`)
- We can also tell if the **SHIFT**, **ALT**, **CONTROL**, or **META** **(Mac)** keys are down at the same time (`isShiftDown()`, `isAltDown()`, …)

Key Events

- Not all keys generate a character we can display (some don't generate a character *at all*!)
- For the `KEY_PRESSED` and `KEY_RELEASED` events, `e.getCode()` tells us which key was pressed or released (whether or not it generates a character)
  - Typically used for non-character-generating keys like the cursor keys (`KeyCode.UP`, `KeyCode.PAGE_DOWN`, etc.). See here for the full list

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.

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

Key Events - Demo

-In the example below:

- If the user types a letter or digit, it replaces the character displayed.
- If the user presses the `UP`, `DOWN`, `LEFT`, or `RIGHT` cursor-movement keys, the letter moves in the corresponding direction by 10 pixels.

Key Events - Demo

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class KeyEventDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    Pane pane = new Pane();
    Text text = new Text(20, 20, "A");

    pane.getChildren().add(text);
    text.setOnKeyPressed(e -> {
      switch (e.getCode()) {
        case DOWN: text.setY(text.getY() + 10); break;
        case UP:  text.setY(text.getY() - 10); break;
        case LEFT: text.setX(text.getX() - 10); break;
        case RIGHT: text.setX(text.getX() + 10); break;
        default:
          if (Character.isLetterOrDigit(e.getText().charAt(0)))
            text.setText(e.getText());
      }
    });

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("KeyEventDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    text.requestFocus(); // text is focused to receive key input
  }
}
```

When our `Text` node receives a `KeyPressed` event, if it's `UP`, `DOWN`, `LEFT`, or `RIGHT`, move the text 10 pixels in the proper direction.

If it's none of those four, but it IS a letter or digit, then change the content of the `Text` node to whatever the key was.

How do we know our `Text` node (as opposed to some *other* node, if we had more on the pane) will receive the key press?

We give it the *focus* (see p. 605)

Key Events

- In the Circle project example, we added key and mouse controls to the enlarge / shrink the Circle
- The original version allowed **ONLY the buttons to enlarge or shrink the circle's radius**
- Now we add the ability to have the "U" and "D" keys to take the radius Up and Down by having the `CirclePane` handle `KeyPress` events
- We also let the left / right mouse buttons do the same thing
- Again, we make sure the `CirclePane` object receives key events by giving it the focus

Key Events

```java
11 public class ControlCircleWithMouseAndKey extends Application {
12   private CirclePane circlePane = new CirclePane();
13
14   @Override // Override the start method in the Application class
15   public void start(Stage primaryStage) {
16     // Hold two buttons in an HBox
17     HBox hBox = new HBox();
18     hBox.setSpacing(10);
19     hBox.setAlignment(Pos.CENTER);
20     Button btEnlarge = new Button("Enlarge");
21     Button btShrink = new Button("Shrink");
22     hBox.getChildren().add(btEnlarge);
23     hBox.getChildren().add(btShrink);
24
25     // Create and register the handler
26     btEnlarge.setOnAction(e -> circlePane.enlarge());
27     btShrink.setOnAction(e -> circlePane.shrink());
28
29     circlePane.setOnMouseClicked(e -> {
30       if (e.getButton() == MouseButton.PRIMARY) {
31         circlePane.enlarge();
32       }
33       else if (e.getButton() == MouseButton.SECONDARY) {
34         circlePane.shrink();
35       }
36     });
37
38     circlePane.setOnKeyPressed(e -> {
39       if (e.getCode() == KeyCode.U) {
40         circlePane.enlarge();
41       }
42       else if (e.getCode() == KeyCode.D) {
43         circlePane.shrink();
44       }
45     });
46
47     BorderPane borderPane = new BorderPane();
48     borderPane.setCenter(circlePane);
49     borderPane.setBottom(hBox);
50     BorderPane.setAlignment(hBox, Pos.CENTER);
51
52     // Create a scene and place it in the stage
53     Scene scene = new Scene(borderPane, 200, 150);
54     primaryStage.setTitle("ControlCircle"); // Set the stage title
55     primaryStage.setScene(scene); // Place the scene in the stage
56     primaryStage.show(); // Display the stage
57
58     circlePane.requestFocus(); // Request focus on circlePane
59   }
60 }
```

Listing 15.9 (pp. 605 – 606) adds key and mouse controls to the enlarge / shrink Circle project

The original version allowed ONLY the buttons to enlarge or shrink the circle's radius

Now we add the ability to have the "U" and "D" keys to take the radius "U"p and "D"own by having the `CirclePane` handle `KeyPress` events:

```java
38     circlePane.setOnKeyPressed(e -> {
39       if (e.getCode() == KeyCode.U) {
40         circlePane.enlarge();
41       }
42       else if (e.getCode() == KeyCode.D) {
43         circlePane.shrink();
44       }
45     });
```

Key Events

```
11 public class ControlCircleWithMouseAndKey extends Application {
12   private CirclePane circlePane = new CirclePane();
13
14   @Override // Override the start method in the Application class
15   public void start(Stage primaryStage) {
16     // Hold two buttons in an HBox
17     HBox hBox = new HBox();
18     hBox.setSpacing(10);
19     hBox.setAlignment(Pos.CENTER);
20     Button btEnlarge = new Button("Enlarge");
21     Button btShrink = new Button("Shrink");
22     hBox.getChildren().add(btEnlarge);
23     hBox.getChildren().add(btShrink);
24
25     // Create and register the handler
26     btEnlarge.setOnAction(e -> circlePane.enlarge());
27     btShrink.setOnAction(e -> circlePane.shrink());
28
29     circlePane.setOnMouseClicked(e -> {
30       if (e.getButton() == MouseButton.PRIMARY) {
31         circlePane.enlarge();
32       }
33       else if (e.getButton() == MouseButton.SECONDARY) {
34         circlePane.shrink();
35       }
36     });
37
38     circlePane.setOnKeyPressed(e -> {
39       if (e.getCode() == KeyCode.U) {
40         circlePane.enlarge();
41       }
42       else if (e.getCode() == KeyCode.D) {
43         circlePane.shrink();
44       }
45     });
46
47     BorderPane borderPane = new BorderPane();
48     borderPane.setCenter(circlePane);
49     borderPane.setBottom(hBox);
50     BorderPane.setAlignment(hBox, Pos.CENTER);
51
52     // Create a scene and place it in the stage
53     Scene scene = new Scene(borderPane, 200, 150);
54     primaryStage.setTitle("ControlCircle"); // Set the stage title
55     primaryStage.setScene(scene); // Place the scene in the stage
56     primaryStage.show(); // Display the stage
57
58     circlePane.requestFocus(); // Request focus on circlePane
59   }
60 }
```

The left and right mouse buttons can now do the same thing:

Again, we make sure the `CirclePane` object receives key events by giving it the focus (l. 58)

```
38     circlePane.setOnKeyPressed(e -> {
39       if (e.getCode() == KeyCode.U) {
40         circlePane.enlarge();
41       }
42       else if (e.getCode() == KeyCode.D) {
43         circlePane.shrink();
44       }
45     });
```

GUI development and JavaFX Basics

BIRZEIT UNIVERSITY