Chapter 15 Event-Driven Programming and Animations

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.
Objectives

- To get a taste of event-driven programming (§15.1).
- To describe events, event sources, and event classes (§15.2).
- To define handler classes, register handler objects with the source object, and write the code to handle events (§15.3).
- To define handler classes using inner classes (§15.4).
- To define handler classes using anonymous inner classes (§15.5).
- To simplify event handling using lambda expressions (§15.6).
- To develop a GUI application for a loan calculator (§15.7).
- To write programs to deal with MouseEvents (§15.8).
- To write programs to deal with KeyEvents (§15.9).
- To create listeners for processing a value change in an observable object (§15.10).
- To use the Animation, PathTransition, FadeTransition, and Timeline classes to develop animations (§15.11).
- To develop an animation for simulating a bouncing ball (§15.12).

Procedural vs. Event-Driven Programming

- Procedural programming is executed in procedural order
- In event-driven programming, code is executed upon activation of events. We can write code to process events such as:
  - Button clicks
  - Mouse movements
  - Keystrokes
  - Clock events
Events and Event Sources

- An event can be defined as a type of signal to the program that something has happened.
- Event are generated by external user actions such as mouse movements, mouse clicks, or keystrokes. Events can also be generated by a computer’s clock.
- When an event is detected at an event source, an event object is created. This event object is then passed on to an event handler which takes a pre-determined action.

Example: Event-Driven Program

Displays two buttons and display a message on the console when a button is clicked.

1. Extend Application and override start(Stage)
2. Create nodes
3. Create event handlers by implementing EventHandler<> and overriding handle()
4. Register event handlers with event sources (e.g. nodes) using event registration methods
5. Place node in a Parent
6. Place the Parent in the Scene
7. Place the Scene on Stage
8. Show Stage
Detecting & Handling GUI Events

- A source object can detect several types of events
- Several different event handlers may be registered with a single source object to handle different types of events
- When an event occurs, an event object with info on the source object and the specifics of the event is created
- This event object is then handled by the appropriate event handler registered with the source object

Trace Execution

```java
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
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");
    }
}

2. Click OK

3. The JVM invokes the handler's handle method
### Event Classes

```
EventObject → Event → InputEvent
  |                  |
  v                  v
  ActionEvent       MouseEvent
    |              |
    v              v
    KeyEvent      WindowEvent

JavaFX event classes are in the `javafx.event` package.
```

### Contents of the Event Object

- An event object contains whatever properties are pertinent to the event
  - For example, we can identify the source object of the event using `getSource()` in the `EventObject` class

- Subclasses of `EventObject` deal with special types of events, such as button actions, window events, mouse movements, and keystrokes
### The Delegation Model

Java uses a delegation-based model for event handling: a source object fires an event, and a handler (or listener) object interested in the event handles it, i.e. the source object delegates the handling of the event to the handler object.
Example: The Delegation Model

```java
Button btOK = new Button("OK");
OKHandlerClass handler = new OKHandlerClass();
btOK.setOnAction(handler); //register listener
```

Source

Example: Event Handling

We would like to use buttons to control the size of a circle
Inner Class Listeners

- A listener (or handler) class is designed specifically to create a listener object for a GUI component (e.g., a button)

- As it will not be shared by other applications, it is appropriate to define the listener class inside the main class as an inner class

Inner Classes

- An inner class, or nested class, is a class defined within the scope of another class. In some applications, inner classes make the program easier to understand

- 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
Example: Inner Classes

(a) 
```java
public class Test {
  ...
}
public class A {
  ...
}
```

(b) 
```java
public class Test {
  ...
  // Inner class
  public class A {
    ...
  }
}
```

(c) 
```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 m() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

Inner Classes (cont.)

- An inner class supports the work of its outer class and is compiled into a class named `OuterClassName$InnerClassName.class`
  - Example: inner class B in outer class A is compiles to `A$B.class`

- An inner class can be declared subject to the same visibility rules applied to any other member of a class

- An inner class can be declared `static`, but it cannot access non-static members of the outer class. A `static` inner class can be accessed using the outer class name
Anonymous Inner Classes (AIC)

- They allow us to declare & instantiate a class all at once which makes the code more concise
- They are like inner classes except that they do not have a name
- We use them if an inner class needs to be used only once
- AIC declaration:
  
  ```java
  new SuperClassName/InterfaceName() {
    // Implement or override methods in superclass or interface
    // Other methods if necessary
  }
  ```
- Inner class listeners can be shortened using AICs

Anonymous Inner Classes (cont.)

- 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 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()
- 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, these two classes are compiled into Test$1$.class and Test$2$.class
Example: Anonymous Inner Class

(a) Inner class EnlargeHandler

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

(b) Anonymous inner class

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

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All rights reserved.

Event Handling: Lambda Expressions

- Lambda expressions can be used to greatly simplify coding for event handling
- A Lambda expression can be viewed as an anonymous method with a concise syntax
  - For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines

(a) Anonymous inner class event handler

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

(b) Lambda expression event handler

```java
btEnlarge.setOnAction(e -> { 
    // Code for processing event e 
});
```
Lambda Expressions: Basic Syntax

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

or

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

The data type for a parameter may be explicitly declared or implicitly inferred by the compiler. The parentheses can be omitted if there is only one parameter without an explicit data type. For example,

(ActionEvent e) -> { // Code for handling event e }

can be shortened as:

e -> { // Code for handling event e }

EventHandler is a SAM Interface

- The compiler treats a lambda expression as an object created from an anonymous inner class. Therefore, the compiler understands that the object must be an instance of EventHandler<ActionEvent>

- Since the EventHandler interface defines handle() with a parameter of the ActionEvent type, the compiler knows that e is of ActionEvent type, and the statements in the lambda expression are all for the body of handle()

- EventHandler contains just one abstract method. Such interfaces are known as functional or Single Abstract Method (SAM) interfaces
Examples: Lambda Expressions

AnonymousHandlerDemo discussed a little while back can be simplified using lambda expressions as in LambdaHandlerDemo

Another example:

The MouseEvent Class

- A MouseEvent is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene.
- The MouseEvent object captures the event, such as the number of clicks, the location, or which button was pressed.
The **KeyEvent** Class

- A **KeyEvent** is **fired** whenever a key is pressed, released, or typed on a node or a scene.
- The **KeyEvent** object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the key value.

### The **KeyCode** Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOME</td>
<td>The Home key</td>
<td>CONTROL</td>
<td>The Control key</td>
</tr>
<tr>
<td>END</td>
<td>The End key</td>
<td>SHIFT</td>
<td>The Shift key</td>
</tr>
<tr>
<td>PAGE_UP</td>
<td>The Page Up key</td>
<td>BACK_SPACE</td>
<td>The Backspace key</td>
</tr>
<tr>
<td>PAGE_DOWN</td>
<td>The Page Down key</td>
<td>CAPS</td>
<td>The Caps Lock key</td>
</tr>
<tr>
<td>UP</td>
<td>The up-arrow key</td>
<td>NUM_LOCK</td>
<td>The Num Lock key</td>
</tr>
<tr>
<td>DOWN</td>
<td>The down-arrow key</td>
<td>ENTER</td>
<td>The Enter key</td>
</tr>
<tr>
<td>LEFT</td>
<td>The left-arrow key</td>
<td>UNDEFINED</td>
<td>The <strong>keyCode</strong> unknown</td>
</tr>
<tr>
<td>RIGHT</td>
<td>The right-arrow key</td>
<td>F1 to F12</td>
<td>The function keys from F1 to F12</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>The Esc key</td>
<td>0 to 9</td>
<td>The number keys from 0 to 9</td>
</tr>
<tr>
<td>TAB</td>
<td>The Tab key</td>
<td>A to Z</td>
<td>The letter keys from A to Z</td>
</tr>
</tbody>
</table>
Example: ControlCircle w. Mouse & Key

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, for adding a listener, contains addListener(InvalidationListener listener)

- Once the value is changed in the property, a listener is notified. The listener class should implement the InvalidationListener interface, which uses invalidated(Observable o) to handle the property value change

- Every binding property is an instance of Observable
Animation

Animation class and its concrete subclasses (especially PathTransition, FadeTransition and Timeline) provide the core functionality for all animations.

```java
javafx.animation.Animation
- autoReverse: BooleanProperty
- cycleCount: IntegerProperty
- rate: DoubleProperty
- status: ReadOnlyObjectProperty<Animation.Status>

+ pause(): void
+ play(): void
+ stop(): void
```

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

- Defines whether the animation reverses direction on alternating cycles.
- Defines the number of cycles in this animation.
- Defines the speed and direction for this animation.
- Read-only property to indicate the status of the animation.

- Pauses the animation.
- Plays the animation from the current position.
- Stops the animation and resets the animation.

PathTransition

The PathTransition class animates the moves of a node along a path from one end to the other over a given time.

```java
javafx.animation.PathTransition
- duration: ObjectProperty<Duration>
- node: ObjectProperty<Node>
- orientation: ObjectProperty<PathTransition.OrientationType>
- path: ObjectProperty<Shape>

+ PathTransition()
+ PathTransition(duration: Duration, path: Shape)
+ PathTransition(duration: Duration, path: Shape, node: Node)
```

The duration of this transition.
The target node of this transition.
The orientation of the node along the path.
The shape whose outline is used as a path to animate the node move.

- Creates an empty PathTransition.
- Creates a PathTransition with the specified duration and path.
- Creates a PathTransition with the specified duration, path, and node.
FadeTransition

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

```
javafx.animation.FadeTransition
- duration: ObjectProperty<Duration>
- node: ObjectProperty<Node>
- fromValue: DoubleProperty
- toValue: DoubleProperty
- byValue: DoubleProperty
+ FadeTransition()
+ FadeTransition(duration: Duration)
+ FadeTransition(duration: Duration, node: Node)
```

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

The duration of this transition.
The target node of this transition.
The start opacity for this animation.
The stop opacity for this animation.
The incremental value on the opacity for this animation.

Creates an empty FadeTransition.
Creates a FadeTransition with the specified duration.
Creates a FadeTransition with the specified duration and node.

Timeline

- PathTransition and FadeTransition define specialized animations
- The Timeline class can be used to program any animation using one or more KeyFrames
- Each KeyFrame is executed sequentially at a specified time interval
Case Study: Clock Animation

Use the static clock from the last chapter to create a clock animation.

Note: In all of our examples so far, events have been generated by the user. The events for this animation, however, are generated by system clock.

Case Study: Bouncing Ball

Use the BallPane class to create a bouncing ball animation.

declare a BounceBallControl variable
in the Application class
create a BallPane object
set the initial position and size of the ball
set the speed and animation
move the ball
"