**CHAPTER15: EVENT-DRIVEN PROGRAMMING AND ANIMATIONS**

You can write code to process events such as a button click, mouse movement, and keystrokes.

You have to use event-driven programming to write the code to respond to the button-clicking event.

* 1. **EVENTS AND EVENTS SOURCES**

An event is an object created from an event source. Firing an event means to create an event and delegate the handler to handle the event

When you run a Java GUI program, the program interacts with the user and the events drive its execution. This is called **event-driven programming**.

An event can be defined as a signal to the program that something has happened.

Events are triggered by external user actions, such as mouse movements, mouse clicks, and keystrokes.

The program can choose to respond to or ignore an event.

The component that creates an event and fires it is called the **event source object**, or simply **source object** or **source component**.

An event object contains whatever properties are pertinent to the event.

**Table:** **User Action, Source Object, Event Type, Handler Interface, and Handler**

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

* 1. **Registering Handlers and Handling Events**

A handler is an object that must be registered with an event source object and it must be an instance of an appropriate event-handling interface.

Java uses a delegation-based model for event handling: A source object fires an event, and an object interested in the event handles it.

The latter object is called an event handler or an event listener.

For an object to be a handler for an event on a source object, two things are needed:

1. The handler object must be an instance of the corresponding event–handler interface to ensure the handler has the correct method for processing the event. JavaFX defines a unified handler interface **EventHandler<T extends Event>** for an event **T**.
2. The handler object must be registered by the source object. Registration methods depend on the event type. For **ActionEvent**, the method is **setOnAction**. For a mouse-pressed event, the method is **setOnMousePressed**. For a key-pressed event, the method is **setOnKeyPressed**.
   1. **INNER CLASSES**

An inner class, or nested class, is a class defined within the scope of another class. Inner classes are useful for defining handler classes.

An inner class is defined as a member of another class:

**OuterClass {**

**private int data;**

**/\*\* A method in the outer class \*/**

**public void m() {**

**...**

**}**

**class InnerClass {**

**/\*\* A method in the inner class \*/**

**public void mi() {**

**// Directly reference data and method**

**// defined in its outer class**

**data++;**

**m();**

**}**

**}**

**}**

**public class Test {**

**...**

**// Inner class**

**public class A {**

**...**

**}**

**}**

**public class Test {**

**...**

**}**

**public class A {**

**...**

**}**

Normally, you define a class as an inner class if it is used only by its outer class. An inner class has the following features:

* An inner class is compiled into a class named **OuterClassName$InnerClassName.class**.
* An inner class can reference the data and the methods defined in the outer class in which it nests, so you need not pass the reference of an object of the outer class to the constructor of the inner class. For this reason, inner classes can make programs simple and concise.
* An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class.
* An inner class can be defined as **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.
* Objects of an inner class are often created in the outer class. However, you can also create an object of an inner class from another class. If the inner class is nonstatic, you must first create an instance of the outer class, then use the following syntax to create an object for the inner class:

**OuterClass.InnerClass innerObject = outerObject.new InnerClass();**

* If the inner class is static, use the following syntax to create an object for it:

**OuterClass.InnerClass innerObject = new OuterClass.InnerClass();**

* 1. **Anonymous Inner-Class Handlers**

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.

The syntax for an anonymous inner class is shown below:

**new SuperClassName/InterfaceName() {**

**// Implement or override methods in superclass or interface**

**// Other methods if necessary**

**}**

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

* An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit **extends** or **implements** clause.
* An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
* An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is **Object().**
* An anonymous inner class is compiled into a class named **OuterClassName$n.class**.
  1. **Simplifying Event Handling Using Lambda Expressions**

Lambda expressions can be used to greatly simplify coding for event handling.

Lambda expressions can be viewed as an anonymous class with a concise syntax.

The basic syntax for a lambda expression is either:

**(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. The curly braces can be omitted if there is only one statement.

The compiler treats a lambda expression as if it is an object created from an anonymous inner class. The compiler processes a lambda expression in three steps: (1) identify the lambda expression type, (2) identify the parameter types, and (3) identify statements.

Consider the following lambda expression:

**btEnlarge.setOnAction(**

**e −> { // Code for processing event e**

**}**

**);**

It is processed as follows:

* Step 1: The compiler recognizes that the object must be an instance of EventHandler, since the expression is an argument of the setOnAction method.
* Step 2: Since the EventHandler interface defines the handle method with a parameter of the ActionEvent type, the compiler recognizes that e is a parameter of the ActionEvent type.
* Step 3: The compiler recognizes that the code for processing e is the statements in the body of the handle method.

The EventHandler interface contains just one method named handle. The statements in the lambda expression are all for that method. If it contains multiple methods, the compiler will not be able to compile the lambda expression. Therefore, for the compiler to understand lambda expressions, the interface must contain exactly one abstract method. Such an interface is known as a Single Abstract Method (SAM) interface.

In essence, a lambda expression creates an object and the object performs a function by invoking this single method. Thus, a SAM interface is also known as a functional interface, and an instance of a functional interface is known as a function object. Since a lambda expression is squarely on defining a function, a lambda expression is also called a lambda function. The terms lambda expression and lambda function are interchangeable.

* 1. **Mouse Events**

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 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. |

The **MouseEvent** class encapsulates information for mouse events.

Four constants—PRIMARY, SECONDARY, MIDDLE, and NONE—are defined in MouseButton to indicate the left, right, middle, and none mouse buttons, respectively. You can use the getButton() method to detect which button is pressed.

* 1. **Key Events**

A KeyEvent is fired whenever a key is pressed, released, or typed on a node or a scene.

Key events enable the use of the keys to control and perform actions, or get input from the keyboard. The KeyEvent object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key.

|  |  |
| --- | --- |
| 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 KeyEvent class encapsulates information about key events.

The key pressed handler is invoked when a key is pressed, the key released handler is invoked when a key is released, and the key typed handler is invoked when a Unicode character is entered.

Every key event has an associated code that is returned by the getCode() method in KeyEvent. The key codes are constants defined in KeyCode.

For use of enum types, see Appendix I. For the key-pressed and key-released events, getCode() returns the value as defined in the table, getText() returns a string that describes the key code, and getCharacter() returns an empty string.

For the key-typed event, getCode() returns UNDEFINED and getCharacter() returns the Unicode character or a sequence of characters associated with the key-typed event.

**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 |

* 1. **Listeners for Observable Objects**

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

An instance of Observable is known as an observable object, which contains the addListener(InvalidationListener listener) method for adding a listener. The listener class must implement the functional interface InvalidationListener to override the invalidated(Observable o) method for handling the value change. Once the value is changed in the Observable object, the listener is notified by invoking its invalidated(Observable o) method. Every binding property is an instance of Observable.

* 1. **Animation**

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

The abstract Animation class provides the core functionalities for animations in JavaFX,

Many concrete subclasses of Animation are provided in JavaFX.

This section introduces PathTransition, FadeTransition, and Timeline.

|  |  |
| --- | --- |
| javafx.animation.Animation | |
| –autoReverse: BooleanProperty | Defines whether the animation reverses direction on alternating cycles. |
| –cycleCount: IntegerProperty | Defines the number of cycles in this animation. |
| –rate: DoubleProperty | Defines the speed and direction for this animation. |
| –status: ReadOnlyObjectProperty  <Animation.Status> | Read-only property to indicate the status of the animation. |
| +pause(): void | Pauses the animation. |
| +play(): void | Plays the animation from the current position. |
| +stop(): void | Stops the animation and resets the animation. |

The abstract Animation class is the root class for JavaFX animations.

The autoReverse is a Boolean property that indicates whether an animation will reverse its direction on the next cycle.

The cycleCount indicates the number of the cycles for the animation. You can use the constant Timeline.INDEFINITE to indicate an indefinite number of cycles.

The rate defines the speed of the animation. A negative rate value indicates the opposite direction for the animation.

The status is a read-only property that indicates the status of the animation (Animation.Status.PAUSED, Animation.Status.RUNNING, and Animation.Status.STOPPED).

The methods pause(), play(), and stop() pause, play, and stop an animation, respectively.

* + 1. **PathTransition**

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

The UML class diagram for the class is shown below:

|  |  |
| --- | --- |
| javafx.animation.PathTransition | |
| –duration: ObjectProperty<Duration> | The duration of this transition. |
| –node: ObjectProperty<Node> | The target node of this transition. |
| –orientation: ObjectProperty  <PathTransition.OrientationType> | The orientation of the node along the path. |
| –path: ObjectType<Shape> | The shape whose outline is used as a path to animate the node move. |
| +PathTransition() | Creates an empty PathTransition. |
| +PathTransition(duration: Duration, path: Shape) | Creates a PathTransition with the specified duration and path. |
| +PathTransition(duration: Duration, path: Shape, node: Node) | Creates a PathTransition with the specified duration, path, and node. |

The **PathTransition** class defines an animation for a node along a path.

The Duration class defines a duration of time. It is an immutable class. The class defines constants INDEFINITE, ONE, UNKNOWN, and ZERO to represent an indefinte duration, one millisecond, unknown, and zero duration, respectively.

You can use new Duration(double millis) to create an instance of Duration, the add, subtract, multiply, and divide methods to perform arithmetic operations, and the toHours(), toMinutes(), toSeconds(), and toMillis() to return the number of hours, minutes, seconds, and milliseconds in this duration, respectively.

You can also use compareTo to compare two durations.

The constants NONE and ORTHOGONAL\_TO\_TANGENT are defined in PathTransition .OrientationType. The latter specifies that the node is kept perpendicular to the path’s tangent along the geometric path.

* + 1. **FadeTransition**

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

The UML class diagram for the class is shown below:

|  |  |
| --- | --- |
| javafx.animation.FadeTransition | |
| –duration: ObjectProperty<Duration> | The duration of this transition. |
| –node: ObjectProperty<Node> | The target node of this transition. |
| –fromValue: DoubleProperty | The start opacity for this animation. |
| –toValue: DoubleProperty | The stop opacity for this animation. |
| –byValue: DoubleProperty | The incremental value on the opacity for this animation. |
| +FadeTransition() | Creates an empty FadeTransition. |
| +FadeTransition(duration: Duration) | Creates a FadeTransition with the specified duration. |
| +FadeTransition(duration: Duration, node: Node) | Creates a FadeTransition with the specified duration and node. |

The **FadeTransition** class defines an animation for the change of opacity in a node.

* + 1. **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. Timeline inherits from Animation. You can construct a Timeline using the constructor new Timeline(KeyFrame...keyframes).

A KeyFrame can be constructed using:

**new KeyFrame(Duration duration, EventHandler<ActionEvent> onFinished)**

The handler onFinished is called when the duration for the key frame is elapsed.

**THE END!**