Handling Events in JavaFX

## Event

An event is an action or occurrence that happens during the execution of a program. Common examples of events include:

* User clicks a button
* User presses a key
* User moves the mouse

## Event Source

When an event occurs, it is associated with a specific GUI control that triggered it. This GUI control is known as the event source. For example, if a user clicks a `Button`, the `Button` is the event source.

## Event Object

When an event occurs, an **event object** is created to encapsulate information about the event. This information is typically stored in the object's fields, and may contain information such as:

* The type of event
* The event source (more on what this is below)
* The event target
* A timestamp for the event

The information held in an event object can be retrieved to determine what should take place in response to the event.

*\*Event objects in JavaFX are typically instantiated implicitly by the JavaFX runtime when an event occurs. Event objects are not usually instantiated directly by the developer.*

Event objects are instances of subclasses of the `Event` class, which is located in the `javafx.event` package. The type of event object that gets instantiated in response to event depends on the type of event that occurred:

|  |  |  |
| --- | --- | --- |
| **Event Type** | **Event Object Type** | **Description** |
| Button click | `ActionEvent` | Occurs when a button is clicked. |
| Mouse click | `MouseEvent` | Occurs when a mouse button is pressed and released on a node. |
| Mouse entry | `MouseEvent` | Occurs when the mouse enters a node. |
| Mouse exit | `MouseEvent` | Occurs when the mouse exits a node. |
| Key press | `KeyEvent` | Occurs when a key is pressed. |
| Key release | `KeyEvent` | Occurs when a key is released. |
| Key typed | `KeyEvent` | Occurs when a key is typed. |
| Text field input | `ActionEvent` | Occurs when `Enter` is pressed in a text field. |
| Scroll | `ScrollEvent` | Occurs when a mouse wheel or trackpad is used to scroll. |

**Event Targets**

The target of an event refers to the **deepest node** in the scene graph that is intended to **handle** the event. The target of an event is determined by the context in which the event occurs.

*\*It may be best to understand what event handlers are and how they work (explained in the next section) and then return to this section after in order to understand it properly.*

To break this down:

* The root node in the scene graph is the "shallowest" node. The more "ancestors" a node has, the "deeper" it is said to be.
* Similar to "stack unwinding" of exceptions, when an event is unhandled by a node in the scene graph, it is passed up to the parent of that node, then the parent of that node, and so on, until the event is handled by a node in the scene graph (or is unhandled by the root node).
* If an event is not handled by any node in the scene graph, then it is said to simply be unhandled, and nothing happens. This does not cause the program to terminate like an unhandled exception would.
* The process of an event moving up through the scene graph until it is handled (or reaches the root unhandled) is called **event bubbling**.

## Event Handlers

Once an event object is generated, it needs to be processed. This is done by an event handler.

**Handling an event involves an implicit call to the `handle` method of an object of a class or descendent thereof (in traditional Java), or a lambda expression (in modern Java), that implements the `EventHandler` interface. This object is specified in the private `onAction` field of the event target (control).**

There are several things that need to be understood before addressing exactly what an event handler is.

### Anonymous Classes

Consider the code below:

interface Greeting{  
 void greet();  
}  
  
Greeting greeting = new Greeting(){  
 public void greet(){  
 System.out.println("Hello, World!");  
 }  
};  
  
greeting.greet();

This code looks very much like the `greeting` object is directly instantiating the `Greeting` interface. However, since interfaces cannot be instantiated, what is happening implicitly is a nameless (anonymous) class that implements the `Greeting` interface is being created by the JVM, and this class is instantiated as the `greeting` object.

The benefits of anonymous classes include:

* **Conciseness**: anonymous classes reduce boilerplate code.
* **Inline Definition**: Since the class is defined where it's used, the code can be easier to read and understand, as the behaviour is immediately visible where it's needed.
* **Avoid Class Proliferation**: anonymous classes help avoid the proliferation of small classes that are only used once. This keeps the codebase smaller and maintainable.

### Functional Interfaces and Lambda Expressions

A functional interface in Java is an interface that contains exactly one abstract method, although it can contain multiple default or static methods (a default method is one where the implementation is defined in an interface, so just like a regular method in a class but defined in an interface). Functional interfaces are often referred to as single abstract method interfaces (SAM interfaces).

Functional interfaces are typically annotated with `@FunctionalInterface`. The compiler will generate an error if the annotated interface does not meet the criteria of a functional interface.

Why is this important?

You may wonder what the significance is of an interface simply having one abstract method. The main purpose of a functional interface (at least, one that was explicitly annotated as such and not just an interface that happens to have a single abstract method) is to facilitate the use of **lambda expressions**.

**A lambda expression is a shorthand, inline expression that implements the abstract method of a functional interface.**

Unlike the earlier example of the implicit instantiation of an anonymous class, when handling lambda expressions, the JVM uses a more efficient mechanism for implementing the abstract method that avoids the need to create an anonymous class. The details of this process are quite complex and delve into bytecode generation, and therefore are beyond the scope of learning JavaFX development.

Lambda Expression Syntax

The basic syntax of a lambda expression is as follows:

(parameters) -> expression

or

(parameters) -> { statements; }

The following is an example of a lambda expression with no parameters:

() -> System.out.println("Hello, World!");

This expression could be assigned to the `run` method of the `Runnable` functional interface, which has a `void` return type.

Below is the definition of the `Runnable` interface:

@FunctionalInterface

public interface Runnable {

void run();

}

Runnable greet = () -> System.out.println("Hello, World!");

The method could then be called using the following call:

greet.run();

The implementation of the abstract method of a functional interface must match the signature declared in the interface - that is, the parameters and return type must match those specified in the interface.

### `EventHandler` Interface

#### Generics Refresher

Since the `EventHandler` interface makes use of generics, here is a brief refresher of what they are and how they work:

* A generic class uses a type parameter (like `<T>`) which defines a type that can be used with its methods and fields.
* The type parameter is specified when an instance of the class is created.
* All methods and fields in the class that use the type parameter will use the type specified at instantiation.

The following is a (heavily abridged) excerpt from the `java.util.List` class for demonstrating this:

public interface List<E> extends Collection<E> {

boolean add(E e);

E get(int index);

E set(int index, E element);

// This interface contains many other methods.

}

Since the generic type `<E>` does not extend any other class, objects of any type can be added to a list. However, they must all be of the same type.

#### `EventHandler<T extends Event>`

The `EventHandler` interface is used to handle events like mouse clicks, key presses, and other user interactions. It is a functional interface that contains a single method, `handle`. The following is the `EventHandler` interface definition from the JavaFX source code:

@FunctionalInterface

public interface EventHandler<T extends Event> extends EventListener {

void handle(T event);

}

* The generic type `T` extends `Event` (an event object), meaning it can handle any subclass of `Event` (that is, any type of event object, such as a `MouseEvent` or `KeyEvent` etc.).
* The `handle` method is called when the event occurs, and it receives the event object as a parameter. This method contains the code to be executed in response to an event taking place.

### `onAction` Field of Control Classes

JavaFX controls have a private field called `onAction`. This field (traditionally) holds an object of a data type implementing the `EventHandler` interface.

Traditionally, this was done by creating an anonymous class implementing the `EventHandler` interface as follows:

Button button = new Button("Hello World");

button.setOnAction(new EventHandler<ActionEvent>(){

@Override

public void handle(ActionEvent event){

System.out.println("Button clicked!");

}

});

The above code does the following:

1. Creates a `Button` object.
2. Calls the `setOnAction` method on that object, and instantiates an event handler of the `EventHandler<ActionEvent>` type as an anonymous class for handling events generated by the button (where the button is the event source).
3. Overrides the `handle` method to print text to the console indicating that the button has been clicked.

When the `button` is clicked, the JVM will implicitly create an event object of type `ActionEvent` for this event. The `handle` method of the `EventHandler` object stored in the `onAction` field of the `button` will be passed this `ActionEvent` object as an argument.

### Setting `onAction` Using Lambda Expressions

The above syntax, which involves creating an anonymous class that implements the `EventHandler` interface, is typically considered verbose and outdated.

Modern Java uses lambda expressions for implementing the `handle` method of the `EventHandler` interface which is considered more efficient and readable.

The syntax for doing so is as follows:

button.setOnAction(event -> System.out.println("Button clicked!"));

Since the `.setOnAction` method of a `Button` type expects an `ActionEvent` argument, this data type can be omitted in the lambda expression. However, you could still explicitly state the data type using the following syntax:

button.setOnAction((ActionEvent event) -> System.out.println("Button clicked!"));

The traditional way of creating anonymous classes for handling events is worth knowing in order to understand what exactly is happening when using lambda expressions, as well as potentially for dealing with legacy code.