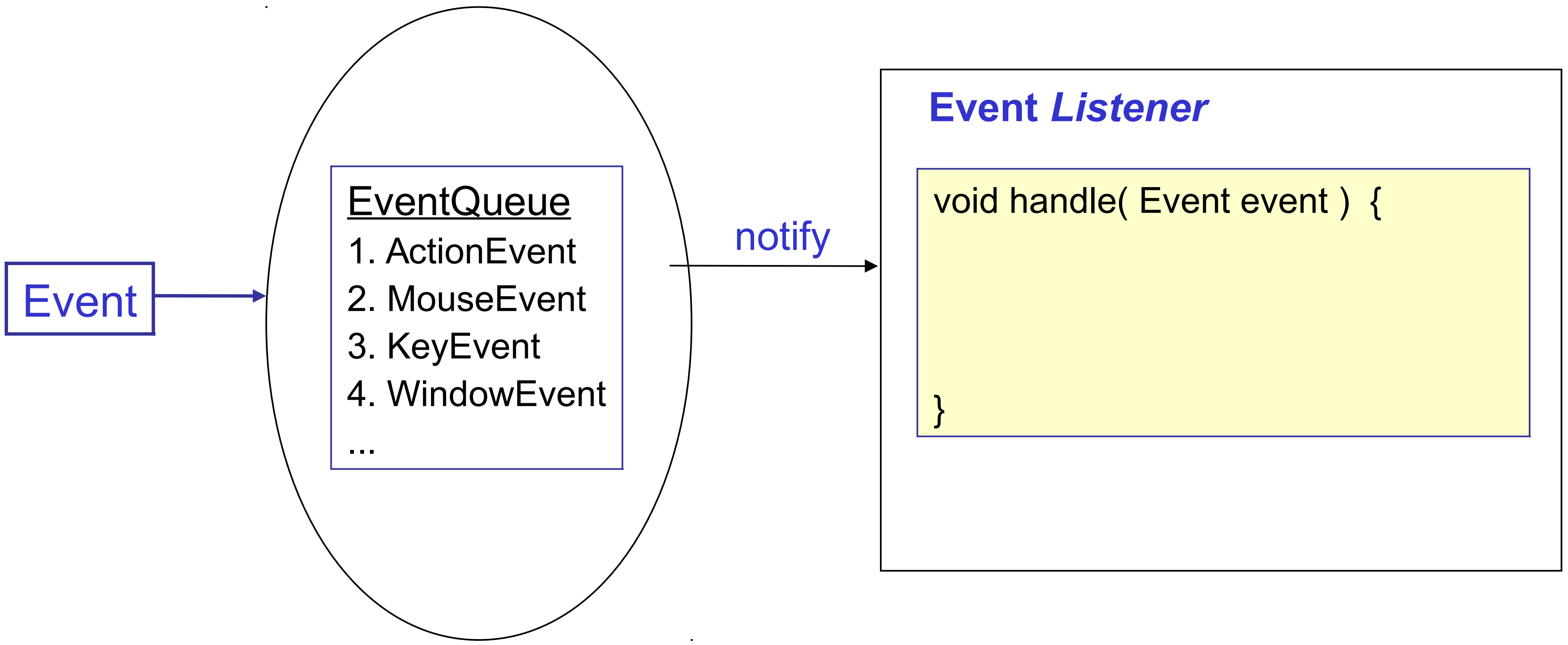
Event Handling in JavaFX

Event Driven Programming

- Graphics applications use ***events***.
- An event dispatcher receives events and ***notifies*** interested objects.



```
1. ActionEvent
2. MouseEvent
3. KeyEvent
4. WindowEvent
...
```

```
void handle( Event event ) {

}
```

Responding to Behavior

Your application must ***do something*** when an event occurs.

Things you need to know

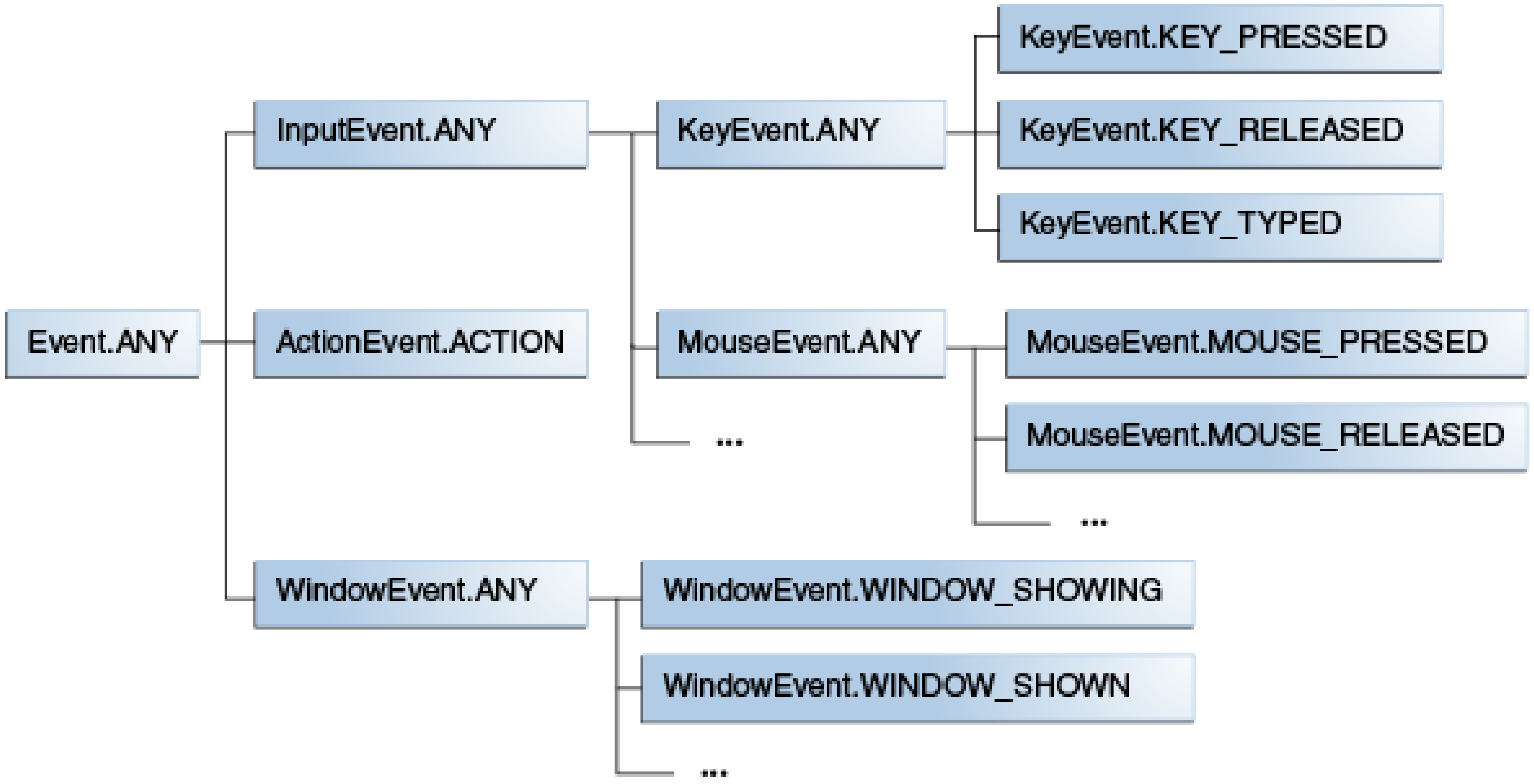
- what kinds of events are there?
- what user (or software) action causes what event?
- how do you write an event handler?
- how do you add event handler to a component?

Check the Event class API

All Events are subclasses of Event.

```
Event
    ActionEvent
    InputEvent
        KeyEvent
        MouseEvent
    WebEvent
    WindowEvent
```

Types of Events



From Oracle's JavaFX Tutorial:

https://docs.oracle.com/javase/8/javafx/events-tutorial/processing.htm

Source of Events

A component or node can be "source" of many kinds of events.

Some event types are different for each node or component.

Its not complicated! Mostly you can *guess* event types.

Button	ActionEvent (button press)
TextField	ActionEvent KeyEvent (Key Press, Key Release, Key Typed).
Any kind of Node	MouseEvent: MousePress, MouseReleased, MouseClicked, MouseDragged, etc. Rotation events, Touch events

What is an EventHandler?

JavaFX uses just *one interface* for all kinds of Event Handlers. This is a lot simpler than Swing and AWT.

<<interface>>

EventHandler<*T extends Event*>

+handle(event: T): void

How to Add Event Handler

There are two ways.

1) **`addEventHandler`** - the general way

2) **`setOnXxxxx`** - convenience methods for specific event type, such as:

```
setOnAction( EventHandler<ActionEvent> e )

setOnKeyTyped( EventHandler<KeyEvent> e )

setOnMouseClicked( EventHander<MouseEvent> e )

setOnMouseMoved( EventHander<MouseEvent> e )
```

...

Example: ENTER or Button click

1. User types his name and clicks a button (or ENTER)

Event type is: **`ActionEvent`**

```
class ButtonHandler
      implements EventHandler<ActionEvent>
{
   public void handle(ActionEvent evt) {
       String text = nameField.getText();
       //TODO greet user using Alert box
       nameField.setText("");  // clear input
    }
}
```

2 Ways to Add Event Handler (demo)

```
// 1. use addEventHandler:
button.addEventHandler(
  ActionEvent.ALL,  new ButtonHandler( ) )

// 2. use setOnAction
button.setOnAction( new ButtonHandler( ) )
```

```
Notice that the EventHandler is the same.
The result will be the same, too.
Both add Event Handler for ActionEvents.
```

You can re-use event handlers

For clarity, or to *reuse* the event handler on multiple components, create the handler first.

```
ButtonHandler greetHandler =
                new ButtonHandler( );
button.setOnAction( greetHandler );
nameField.setOnAction( greetHandler );
```

It is bad programming to create two objects to do the same thing (greet the user).

4 Ways to Define an EventHandler

1. Define an (inner) class that implements EventHandler.
   We just did that.

2. Write it as *anonymous class*.

3. Write it as a *method* and use a *method reference*.
   Method reference is new in Java 8.
   Works because Event Handler has only 1 method.

4. Write it as a *lambda expression* and use a reference variable to add it.

Event Handler as Anonymous Class

You must specify what interface you are implementing, including type parameter.

```
EventHandler<ActionEvent> buttonHandler =
  new EventHandler<ActionEvent>() {
    // anonymous class definition:
    public void handle(ActionEvent evt) {
      String text = nameField.getText();
      //TODO greet user using Alert box
      nameField.setText("");  // clear input
   }
 };
button.setOnAction( buttonHandler );
```

Avoid inline definition & use

This is hard to understand and hard to maintain.
Avoid it. Define the anonymous class *first*, then use it.

```
// This is harder to understand, especially
// when the anonymous class is long.
button.setOnAction(
  new EventHandler<ActionEvent>() {
    public void handle(ActionEvent evt) {
      String text = nameField.getText();
      //TODO greet user using Alert box
      nameField.setText("");  // clear input
   }
  } );
```

Method as Event Handler?

Using SceneBuilder to assign event handlers
we did not write inner classes or anonymous classes.
We just wrote a method, like this:

```
@FXML
public void greetTheUser(ActionEvent evt) {
    String text = nameField.getText();
    //TODO greet user using Alert box
    nameField.setText("");  // clear input
}
```

SceneBuilder let us use a method as Event Handler, instead of object.

How?

Method References

Java 8 allows a *method reference* to be used as something that implements an interface. The syntax is:

`object::methodname`

```
Runnable nike = this::doit; // method reference
nike.run();                 // calls doit()

// this "looks like" a Runnable.run() method
// so we can use it as method reference.
public void doit( ) {
   System.out.println("Just do it.");
}
```

Method Reference as EventHandler

Write a method with the required method signature, but any name you like.

```
// Assign event handler using method reference
  button.setOnAction( this::greetAction );

  // this method signature "looks like" an
  // EventHandler, but the name is different
  public void greetAction(ActionEvent evt) {
    String text = nameField.getText();
    //TODO greet user using Alert box
    nameField.setText("");  // clear input
 }
```

Lambda Expressions

Lambda Expression is an inline method definition, without a method name.

```
EventHandler<ActionEvent> buttonHandler =
    (event) -> {
            String text = nameField.getText();
            //TODO greet user using Alert box
            nameField.setText("");
         } ;
button.setOnAction( buttonHandler );
```

5th Way to Define Event Handler

You can define the controller itself as "implements EventHandler<T>" and use "setOnAction(this)".

```
class GreetController
      implements EventHandler<ActionEvent> {
    @FXML
    public void initialize() {
       button.setOnAction( this );
    ...

    public void handle(ActionEvent event) {
       // handle it.
```

This technique is not usually the best choice. You usually have many components which need custom event handlers.

Event Handling Exercise

- Draw a Sequence Diagram of logic for creating and using an ActionEvent handler.

Event Dispatching

When an event occurs, JavaFX does:

1. Determine the event **target**.

2. Event Capture: pass the event down from the root node to the target.

Along the way, EventFilters may be invoked.

3. Event Handling (Event Bubbling): starting at the target, any event handler is invoked. The event "bubbles" back up the tree until it is consumed.

See:

https://www.tutorialspoint.com/javafx/javafx_event_handling.htm

References

Event Handling in Oracle *JavaFX Tutorial.* *This has the most complete explanation of event types and event handling*

https://docs.oracle.com/javase/8/javafx/events-tutorial/events.htm

Event Handling in *Tutorialspoint JavaFX Tutorial.*

https://www.tutorialspoint.com/javafx/javafx_event_handling.htm - example of event capture, event filter, and event handler.

JavaFX Events

http://zetcode.com/gui/javafx/events/