Programming of Interactive Systems

Anastasia.Bezerianos@lri.fr

Housekeeping & reminders

Sent feedback on your Minesweeper storyboard (individual)

(any missing? we'll take some time to chat)

Your Maze storyboard is due today (group) !!!

Your Minesweeper v1 is due next Mon

Homework-2

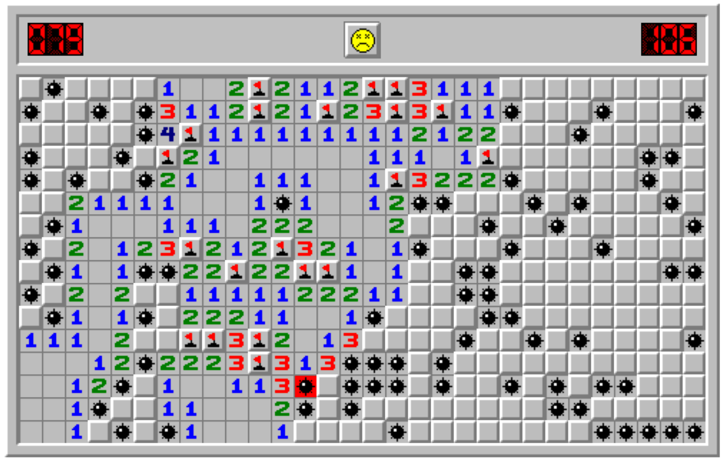
Minesweeper v1: due on **Sep 27th**

upload to campus
(or send to: introduction.prog.is@gmail.com)

see announcement for details:

- code the layout of your windows
- code your UI elements
  (put buttons and labels in cells for now)
- add one reaction to an input
  (e.g., remove a button to
  show a label)

You'll **DEMO** next week in class



Today's class is one of the most challenging (and useful !!!) in the class

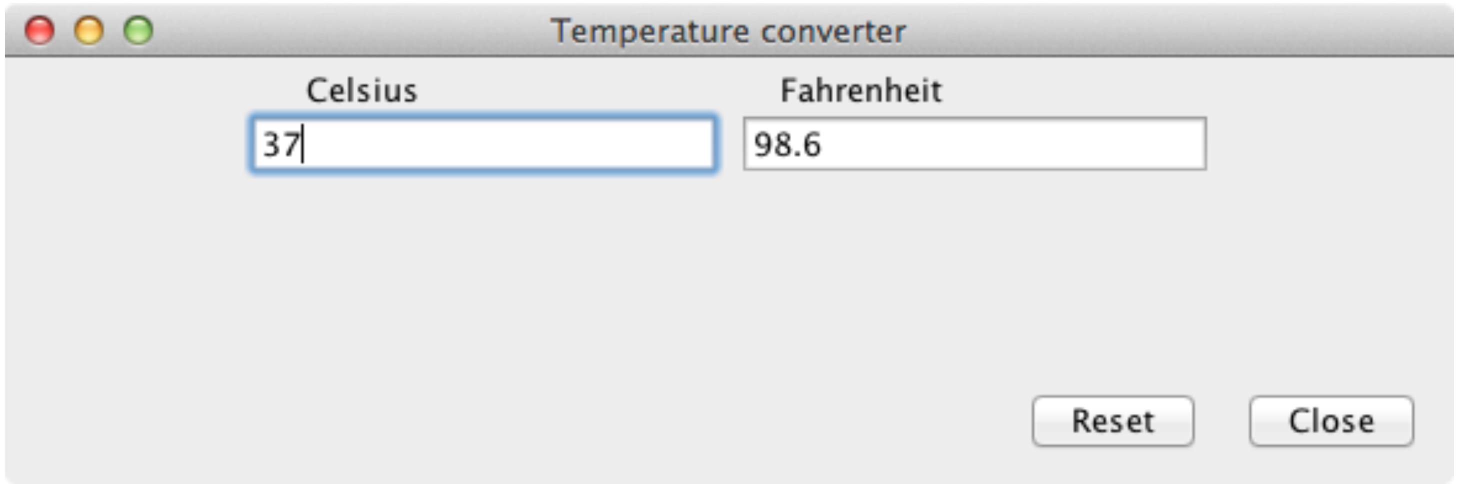
follow class, interrupt and ask questions!!!
re-watch and re-read slides, come to your TAs for more questions

Week 3 :
a. JavaFX user input

Anastasia.Bezerianos@lri.fr

(part of this class is based on previous classes from Anastasia,
and of T. Tsandilas, S. Huot, M. Beaudouin-Lafon, N.Roussel, O.Chapuis)

user input


Temperature converter
Celsius
37
Fahrenheit
98.6
Reset
Close

event driven programming

You may have done structured sequential programming where the order of execution is controlled by the programmer.

GUIs use **event-driven** programming:

- User is presented with options.
- User actions (and other actions) fire events.
- Event handlers execute in response to events.
- Order of execution is controlled by the order of events, which the programmer does not know in advance.

event handlers (JavaFX)

Event: A msg (in Java an object) that represents a user's interaction with a GUI component; can be "handled" to create interactive components

Handler (Listener*): A method or object that waits for events and responds to them.

– To handle an event, we attach a *handler* to a UI component.
– The handler(listener) will be notified by the system when the event occurs (e.g., button click)

(*) If you've used Swing before, Handlers are classes with factory methods for creating Listeners

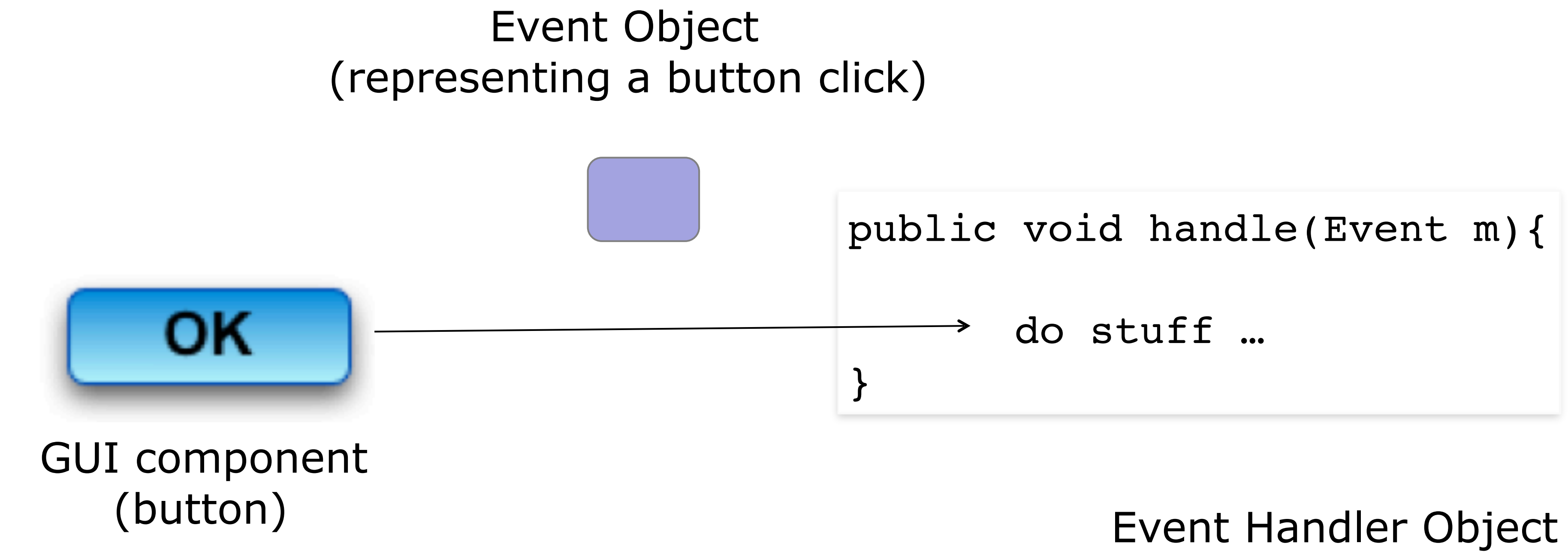
event handlers (JavaFX)

methods of type `setOnMouseEnter()` or `addEventHandler()` create the **handler** object that treats events

when the system recognises an input event (e.g., click on an object), it triggers the predefined method `handle()`for that event

Java has many many ways of dealing with events (AWT,Swing,JavaFX). Only import JavaFX classes

event handlers (Java)



```
public void handle(Event m){

        do stuff …
}
```

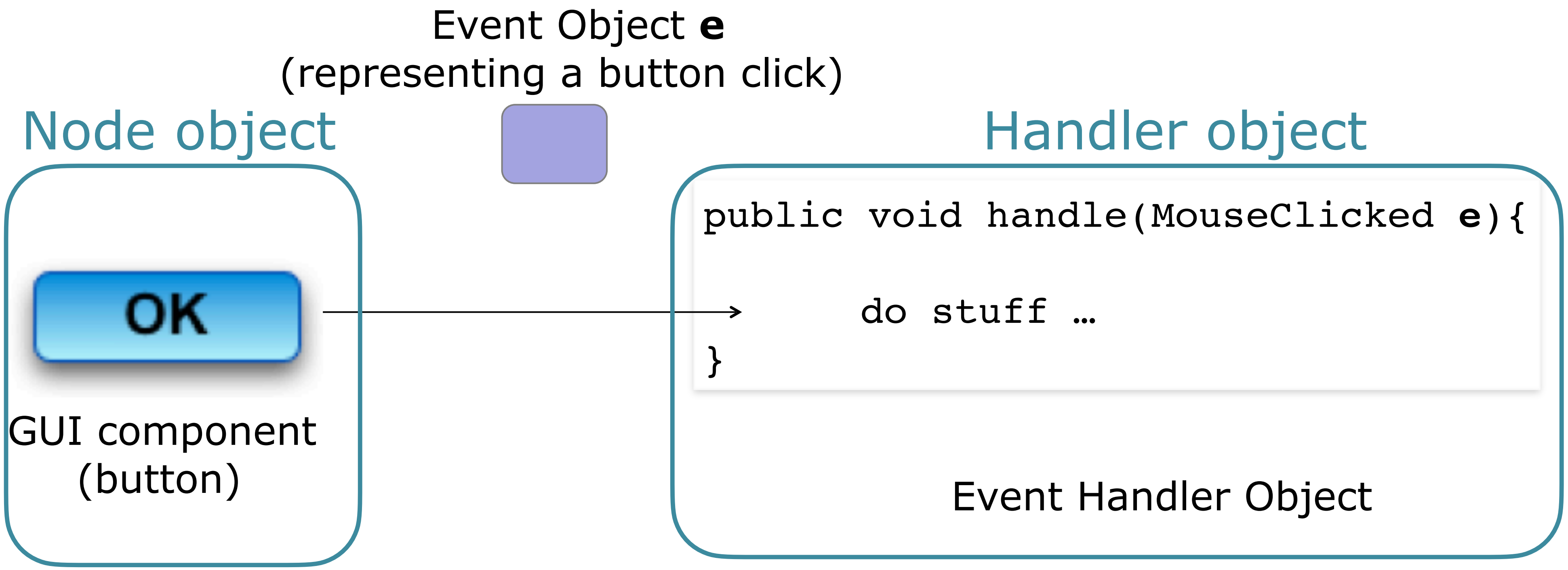
A GUI component sending an event to its “registered" handler

Run configurations > (x)=... under VM arguments put
`--add-modules javafx.controls,javafx.fxml`
& untick "Use the -XstartOnFirstThread

simplest example

```
public class SimplestEvent extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        btn.setOnMouseClicked(e ->{
                    System.out.println("Clicked!");
        });

        root.getChildren().add(btn);
        Scene scene = new Scene(root, 300, 250);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

event handlers (Java)



```
public void handle(MouseClicked e){

        do stuff …
}
```

SetOnMouseClicked … connects the UI node object with a handler object (often we say we “register" the handler to the node) to treat the MouseClicked event

event handlers (JavaFX)

A handler is a Class with code that determines how to process incoming events

A Handler object (instance of that class) is attached to a Node object (e.g., a UI control, like a button)

… but there are many ways to do this !

event handlers (JavaFX)

disclaimer: we start from the most complex, but complete way, and then we are simplifying !!!

event handlers - 1

1st approach:

Separate classes for handlers and UI nodes

event handlers - 1
… 1st class, the handler

```java
package examples.EventExamples;

import javafx.event.EventHandler;
import javafx.event.EventTarget;
import javafx.scene.input.InputEvent;

// the handler class, implements the Interface EventHandler for InputEvent
// but more specific events are also possible
public class MyEventController implements EventHandler<InputEvent>{

@Override
public void handle(InputEvent event) {

          // find the event type, it can be more precise than InputEvent
          String e = event.getClass().getName();

          // find the source (the Node or UI Control) where the event was triggered
          String c = event.getSource().getClass().toString();

          System.out.println("Detected " + e + " on a "+ c);
   }
}
```

cont’d

event handlers - 1
… 1st class, the handler

Our handler class implements an existing Java Interface, `EventHandler`
Notice the <**`event_type`**> syntax
We are forced to implement `handle(Event)`

```java
package examples.EventExamples;

import javafx.event.EventHandler;
import javafx.event.EventTarget;
import javafx.scene.input.InputEvent;

// the handler class, implements the Interface EventHandler for InputEvent
// but more specific events are also possible
public class MyEventController implements EventHandler<InputEvent>{

@Override
public void handle(InputEvent event) {

          // find the event type, it can be more precise than InputEvent
          String e = event.getClass().getName();

          // find the source (the Node or UI Control) where the event was triggered
          String c = event.getSource().getClass().toString();

          System.out.println("Detected " + e + " on a "+ c);
   }
}
```

cont’d

(aside) Java Interface

A Java interface is:

a pattern/structure that creates a

group of related methods with empty bodies

When we *implement* one (or more interfaces)

we need to provide code for its methods

event handlers - 1
… 2nd class, the window using the handler

continuing …

```
package examples.EventExamples;

public class SimpleReactiveButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        // attaching an object of the Handler that we defined (MyEventController)
        // that should treat all possible input events InputEvent.ANY
        btn.addEventHandler(InputEvent.ANY, new MyEventController());

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

event handlers - 1
… 2nd class, the window using the handler

continuing …

```
package examples.EventExamples;

public class SimpleReactiveButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        // attaching an object of the Handler that we defined (MyEventController)
        // that should treat all possible input events InputEvent.ANY
        btn.addEventHandler(InputEvent.ANY, new MyEventController());

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Attaching a handler to our UI control
Parameters: the event to listen for
and the object Handler

event handlers - 1
… the window but with a different event

… or …

```
package examples.EventExamples;

public class SimpleReactiveButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        // attaching an object of the Handler that we defined (MyEventController)
        // that should handle only mouse click events
       btn.addEventHandler(MouseEvent.MOUSE_CLICKED, new MyEventController());

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

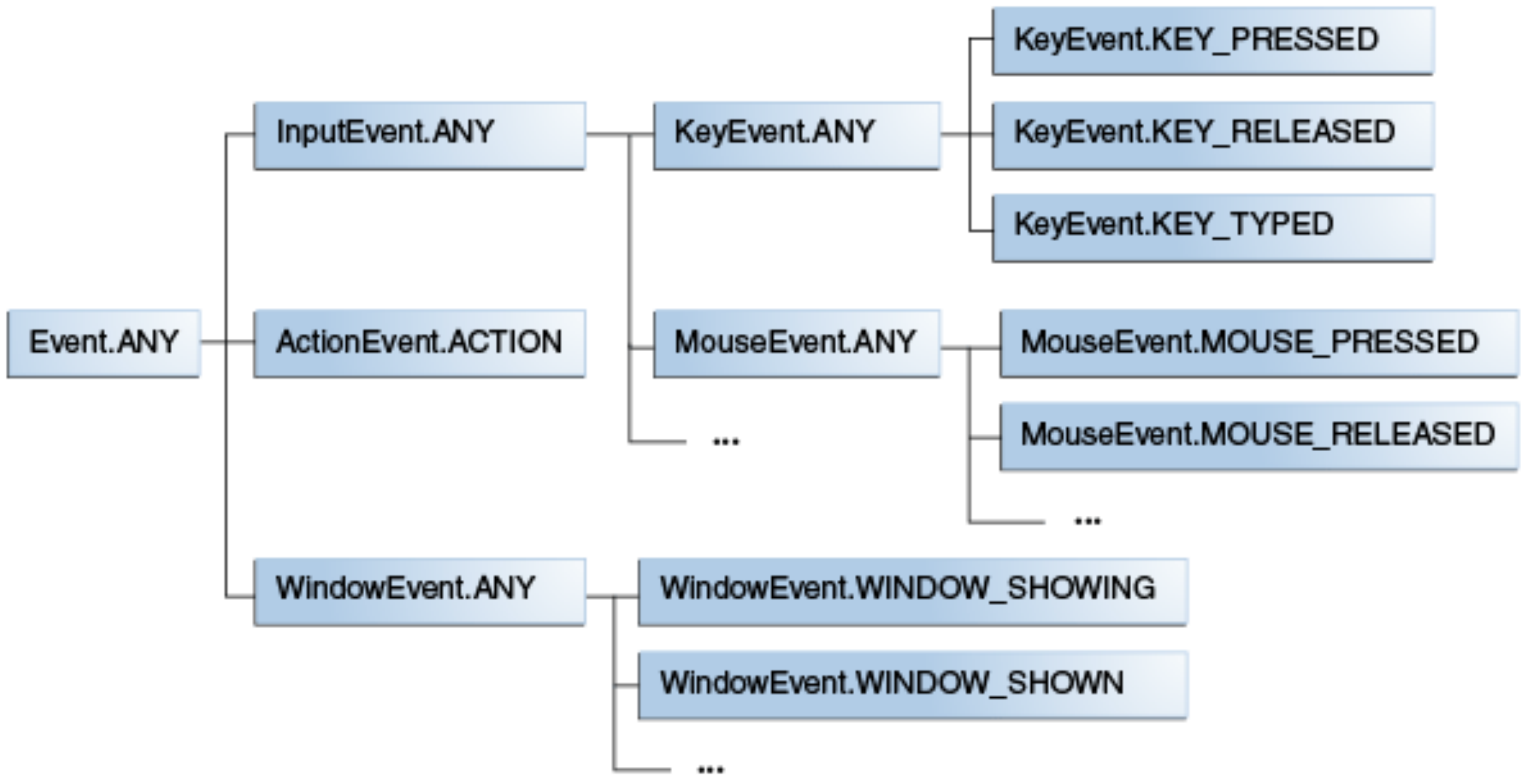
event handlers - 1
… the window but with a different event

… or …

```
package examples.EventExamples;

public class SimpleReactiveButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        // attaching an object of the Handler that we defined (MyEventController)
        // that should handle only mouse click events
       btn.addEventHandler(MouseEvent.MOUSE_CLICKED, new MyEventController());

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Can listen only for a more specific type of event

all events (JavaFX)



https://docs.oracle.com/javafx/2/events/processing.htm for all events and convenience classes

event handlers - 1

1st approach:

Separate classes for handlers and UI nodes

+ code organization (separate input / output)
+ can reuse the handler code (eg if several buttons of your interface that are in different windows do the same thing)
- your handler code is in a different class than your UI nodes, so not all nodes are visible

event handlers - 2

2nd approach:

Handler definition/code at the moment when we attach the handler to a Node (e.g., UI control), in `addEventHandler()`

event handlers - 2
… add and define handler together

```
package examples.EventExamples;

public class StandAloneReactiveButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        // create a handler object and its code all in one go!
        btn.addEventHandler(InputEvent.ANY,
               new EventHandler<InputEvent>() {
                      @Override public void handle(InputEvent e) {
                            System.out.println("Detected " + e.getClass().getName() +
                                       " on a "+ e.getSource().getClass().getName());
                      }
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

event handlers - 2
… add and define handler together

```
package examples.EventExamples;

public class StandAloneReactiveButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        // create a handler object and its code all in one go!
        btn.addEventHandler(InputEvent.ANY,
               new EventHandler<InputEvent>() {
                      @Override public void handle(InputEvent e) {
                            System.out.println("Detected " + e.getClass().getName() +
                                        " on a "+ e.getSource().getClass().getName());
                      }
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Code from our old MyEventController Handler class now exists inside the function `addEventListener()`

Notice the keyword **new**

event handlers - 2
… add and define handler together

```
package examples.EventExamples;

public class StandAloneReactiveButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        // create a handler object and its code all in one go!
        btn.addEventHandler(MouseEvent.MOUSE_CLICKED,
               new EventHandler<InputEvent>() {
                    @Override public void handle(InputEvent e) {
                         System.out.println("Detected " + e.getClass().getName() +
                                   " on a "+ e.getSource().getClass().getName());
                    }
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Can also make sure it handles more precise events

(aside) Java inner classes

Anonymous Inner classes

```
new <class-name> () { <body> };
```

this construction does 2 things:

- creates a new class without name, that is a subclass of `<class-name>` defined by `<body>`
- creates a (unique) instance/object of this new class and returns its value

This (inner) class has access to variables and methods of the class inside which it is defined

(aside) Java inner classes

```java
public class ButtonChangeLabel extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);

        Label lbl = new Label("0");
        Button btn = new Button("Hello World");

        btn.addEventHandler(
                MouseEvent.MOUSE_CLICKED,
              new EventHandler<InputEvent>() {
                    @Override public void handle(InputEvent e) {
                         String str = lbl.getText();
                         int counter = Integer.parseInt(str) + 1;
                         lbl.setText( String.valueOf(counter) );
                    }
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().addAll(lbl,btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

(aside) Java inner classes

```
public class ButtonChangeLabel extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);

        Label lbl = new Label("0");
        Button btn = new Button("Hello World");

        btn.addEventHandler(
                MouseEvent.MOUSE_CLICKED,
               new EventHandler<InputEvent>() {
                     @Override public void handle(InputEvent e) {
                          String str = lbl.getText();
                          int counter = Integer.parseInt(str) + 1;
                          lbl.setText( String.valueOf(counter) );
                     }
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().addAll(lbl,btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Can access and modify `lbl`,
as well as everything else inside the
`ButtonChangeLabel` class

(aside) Java inner classes

```
public class ButtonChangeLabel extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);

        Label lbl = new Label("0");
        Button btn = new Button("Hello World");

        // create a handler object and store a reference to it
        EventHandler<InputEvent> e =
               new EventHandler<InputEvent>() {
                 @Override public void handle(InputEvent e) {
                            String str = lbl.getText();
                            int counter = Integer.parseInt(str) + 1;
                            lbl.setText( String.valueOf(counter) );
                      }
        });

        // and reuse it in the class
       btn.addEventHandler(MouseEvent.MOUSE_CLICKED, e);

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().addAll(lbl,btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

In case you want to keep a reference to your inner class object (handler) to reuse it elsewhere in the class

event handlers - 2

2nd approach:

Handler definition/code at the moment when we attach the handler to a Node (e.g., UI control), in `addEventHandler()`

+ good when handler code is simple
+ access to other parts of the class
- bad code organization
- code can only be reused within the class (if you keep a reference to your object)

event handlers - 3

3rd approach:

- Lambda expressions at the moment when we attach the handler to a Node (removes the declaration of the Event Handler).
- Only works in your Event Handler has one method, i.e., `handle()`

event handlers - 3

e.g.,

```
    new EventHandler<InputEvent>() {
        @Override
         public void handle(InputEvent e){
                 … }
      }
```

becomes

```
e->{…}
```

More on lambda expressions https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#lambda-expressions-in-gui-applications

event handlers - 3
… hide the handler class info with lamda

```
package examples.EventExamples;

public class ConvenienceButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        btn.addEventHandler( MouseEvent.MOUSE_CLICKED, e -> {
                            System.out.println("Detected " + e.getClass().getName() +
                                        " on a "+ e.getSource().getClass().getName());
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

event handlers - 3
… hide the handler class info with lamda

```
package examples.EventExamples;

public class ConvenienceButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        btn.addEventHandler( MouseEvent.MOUSE_CLICKED, e -> {
                           System.out.println("Detected " + e.getClass().getName() +
                                      " on a "+ e.getSource().getClass().getName());
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

no method declaration for handler, just code, with syntax
e->{code}

event handlers - 3

3rd approach:

handler with lamda for

+ good for handlers with a single function (handle)
+ good when handler code is simple
+ access to other parts of the class
- bad code organization
- code can only be reused within the class (if you keep a reference to your object)

event handlers - 1b,2b,3b
… simplified even more

Instead of adding the Event and Handler
`addEventHandler(MouseEvent,Handler)`

we can use Convenience methods, of the form **setOnEvent** whose name encode the Event they can handle, e.g.,
`setOnMouseClick(Handler)`

event handlers - 1b, 2b, 3b
… simplified even more

```
package examples.EventExamples;

public class ConvenienceButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        btn.setOnMouseClicked(e ->{
                            System.out.println("Detected " + e.getClass().getName() +
                                       " on a "+ e.getSource().getClass().getName());
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

event handlers - 1b, 2b, 3b … simplified even more

```
package examples.EventExamples;

public class ConvenienceButton extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

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

        btn.setOnMouseClicked(e ->{
                            System.out.println("Detected " + e.getClass().getName() +
                                      " on a "+ e.getSource().getClass().getName());
        });

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

`setOn<Event>` convenience method encodes the type of event it handles

event handlers (JavaFX)
Summary of How/Why

A handler can be defined/coded:

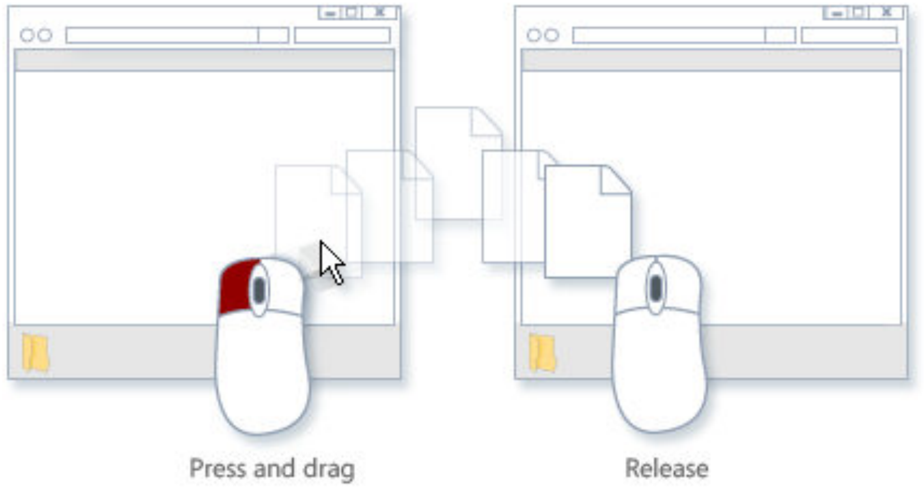
- by adding the handler using lambda expressions `->{…}`
  can only define a `handle` function, simple code specific to that Node, access to other nodes in the class (3)
- by adding the handler using anonymous inner classes
  for simple code specific to that Node
  easily accessing other UI elements (2)
- by using a different class
  for reusable or long/complex code (1)

A handler can be added to a Node using

- convenience methods `setOn<Event>(Handler)`
- method `addEventHandler(Event, Handler)`

« drag-and-drop » to think about

What are the affected « widgets/nodes »?
What are the events?



How to describe this interaction with « event handlers » ?

… JavaFX has drag&drop events

event handlers (JavaFX), more …

handlers need to be registered (added) to nodes

a widget/node can have many handlers

  e.g., one for “click” events and for “enter inside” button events

a handler can be added to multiple nodes

  e.g., one handler (either class or object)

  handles events from multiple buttons

event handlers (JavaFX), more …

handlers need to be registered (added) to nodes

a widget/node can have many handlers

- e.g., one for "click" events and for "enter inside" button events

a handler can be added to multiple nodes

- e.g., one handler (either class or object) handles events from multiple buttons

event details (JavaFX), more …
multiple events

All events have a source, a target, and a type(*)

We will be using source in most cases: `event.getSource()`
which returns an Object of class Object

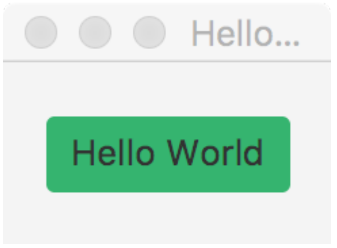
If you <u>know</u> your source you can cast it, e.g.,
`(Button)event.getSource()`
now can be treated as a button

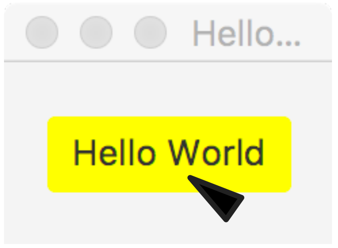
It is best to handle the most specific event you need, e.g.,
`MouseEvent.MOUSE_CLICKED` is more specific than
`InputEvent.ANY`

(*)more on this and on the dispatch chain later

multiple event handlers (JavaFX)

```
public class StyledReactiveButton extends Application {

        @Override
        public void start(Stage primaryStage) {
            primaryStage.setTitle("Hello World");

            FlowPane root = new FlowPane();
            root.setAlignment(Pos.CENTER);

            Button btn = new Button("Hello World");
            btn.setStyle("-fx-background-color: MediumSeaGreen");

            btn.setOnAction(e->{
                    System.out.println("Hello World !!!");
            });
            btn.setOnMouseEntered(e->{
                   btn.setStyle("-fx-background-color: Yellow");
            });
            btn.setOnMouseExited(e->{
                    btn.setStyle("-fx-background-color: MediumSeaGreen");

            });

            Scene scene = new Scene(root, 300, 250);
            root.getChildren().add(btn);
            primaryStage.setScene(scene);
            primaryStage.show();
        }
}
```



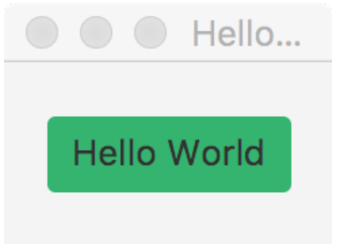


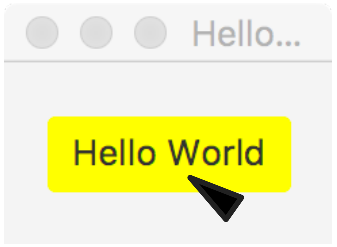
multiple event handlers (JavaFX)

```
public class StyledReactiveButton extends Application {

        @Override
        public void start(Stage primaryStage) {
            primaryStage.setTitle("Hello World");

            FlowPane root = new FlowPane();
            root.setAlignment(Pos.CENTER);

            Button btn = new Button("Hello World");
            btn.setStyle("-fx-background-color: MediumSeaGreen");

            btn.setOnAction(e->{
                    System.out.println("Hello World !!!”);
            });
            btn.setOnMouseEntered(e->{
                    btn.setStyle("-fx-background-color: Yellow");
            });
            btn.setOnMouseExited(e->{
                     btn.setStyle("-fx-background-color: MediumSeaGreen”);

            });

            Scene scene = new Scene(root, 300, 250)
            root.getChildren().add(btn);
            primaryStage.setScene(scene);
            primaryStage.show();
        }
}
```

Use of CSS syntax inline with `setStyle()`, without loading a CSS file

Different event handlers on the same Button



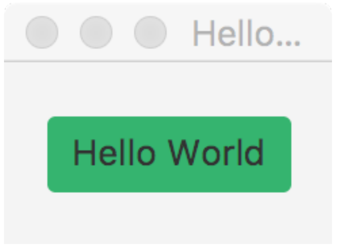


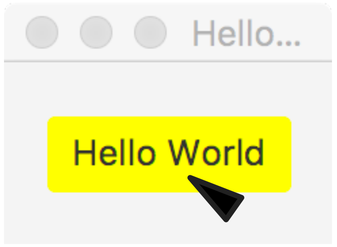
multiple event handlers (JavaFX)

```
public class StyledReactiveButton extends Application {

        @Override
        public void start(Stage primaryStage) {
            primaryStage.setTitle("Hello World");

            FlowPane root = new FlowPane();
            root.setAlignment(Pos.CENTER);

            Button btn = new Button("Hello World");
            btn.setStyle("-fx-background-color: MediumSeaGreen");

            btn.setOnAction(e->{
                    System.out.println("Hello World !!!”);
            });
            btn.setOnMouseEntered(e->{
                    btn.setStyle("-fx-background-color: Yellow");
            });
            btn.setOnMouseExited(e->{
                     btn.setStyle("-fx-background-color: MediumSeaGreen”);

            });

            Scene scene = new Scene(root, 300, 250);
            root.getChildren().add(btn);
            primaryStage.setScene(scene);
            primaryStage.show();
        }
}
```

Use of CSS syntax inline with `setStyle()`, without loading a CSS file

Different event handlers on the same Button



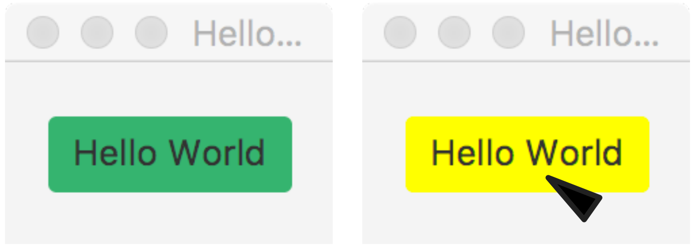


multiple event handlers (JavaFX)

```
public class StyledReactiveButton extends Application {

        @Override
        public void start(Stage primaryStage) {
            primaryStage.setTitle("Hello World");

            FlowPane root = new FlowPane();
            root.setAlignment(Pos.CENTER);

            Button btn = new Button("Hello World");
            btn.setStyle("-fx-background-color: MediumSeaGreen");

            btn.setOnAction(e->{
                    System.out.println("Hello World !!!");
            });
            btn.setOnMouseEntered(e->{
                    Button b = (Button) e.getSource();
                    b.setStyle("-fx-background-color: Yellow");
            });
            btn.setOnMouseExited(e->{
                    Button b = (Button) e.getSource();
                    b.setStyle("-fx-background-color: MediumSeaGreen");
            });

            Scene scene = new Scene(root, 300, 250);
            root.getChildren().add(btn);
            primaryStage.setScene(scene);
            primaryStage.show();
        }
}
```

Instead of accessing btn you can use `(Button) event.getSource()`

to treat the event source as a button …

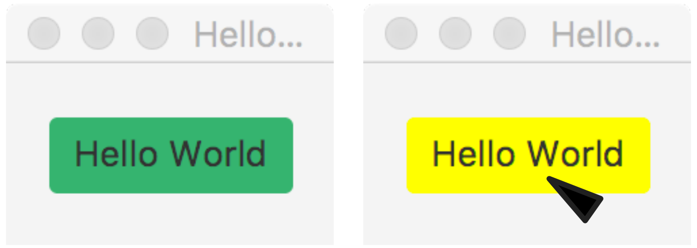


multiple event handlers (JavaFX)

```
public class StyledReactiveButton extends Application {

        @Override
        public void start(Stage primaryStage) {
            primaryStage.setTitle("Hello World");

            FlowPane root = new FlowPane();
            root.setAlignment(Pos.CENTER);

            Button btn = new Button("Hello World");
            btn.setStyle("-fx-background-color: Mediur

            btn.setOnAction(e->{
                    System.out.println("Hello World !!!”);
            });
            btn.setOnMouseEntered(e->{
                      Button b = (Button) e.getSource();
                      b.setStyle("-fx-background-color: Yellow");
            });
            btn.setOnMouseExited(e->{
                      Button b = (Button) e.getSource();
                      b.setStyle("-fx-background-color: MediumSeaGreen”);
            });

            Scene scene = new Scene(root, 300, 250);
            root.getChildren().add(btn);
            primaryStage.setScene(scene);
            primaryStage.show();
        }
}
```

…

This can be used in handlers that are outside this class (to access the source UI element)



event handlers (JavaFX), more …

handlers need to be registered (added) to nodes

a widget/node can have many handlers

- e.g., one for “click” events and for “enter inside” button events

a handler can be added to multiple nodes

- e.g., one handler (either class or object) handles events from multiple buttons

event handlers (JavaFX), more …

handlers need to be registered (added) to nodes

a widget/node can have many handlers

- e.g., one for “click” events and for “enter inside” button events

a handler can be added to multiple nodes

- e.g., one handler (either class or object) handles events from multiple buttons

event handlers (JavaFX), more …
1.sharing handlers

```
package examples.EventExamples.separate;


import javafx.event.EventHandler;
import javafx.event.EventTarget;
import javafx.scene.control.Button;
import javafx.scene.input.InputEvent;

public class MySecondEventController implements EventHandler<InputEvent> {

        int counter = 0;

        @Override
        public void handle(InputEvent event) {
            // find the event type, it can be more precise than InputEvent
                    String e = event.getClass().getName();

            // find the source (the Node or UI Control) where the event was triggered
                    String c = event.getSource().getClass().getName();

                    Button b = (Button) (event.getSource() );
                    String n = b.getText();


                    System.out.println( counter +": Detected " + e + " on a "+ c + " with text " + n );

                    ++counter;
        }
}
```

event handlers (JavaFX) more …
1a. sharing handlers as a *class*

```
package examples.EventExamples;

public class ManySimpleReactiveButtons extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);
         Button btn1 = new Button("Hello World");
         Button btn2 = new Button("Hello Again");


        // Creating a new handler object for each button
       // each button has their own Handler object (run a different copy of the code)
        btn1.addEventHandler(InputEvent.ANY, new MySecondEventController());
        btn2.addEventHandler(InputEvent.ANY, new MySecondEventController());

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().addAll(btn1,btn2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

event handlers (JavaFX) more …
1b. sharing handlers as *objects (advanced)*

```
package examples.EventExamples;

public class ManySimpleReactiveButtons extends Application {

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World");

        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);
        Button btn1 = new Button("Hello World");
        Button btn2 = new Button("Hello Again");


        // Creating a single Handler object
       MySecondEventController myhandler = new MySecondEventController();
       // Adding the same handler object to both buttons, they run the exact same code
        btn1.addEventHandler(InputEvent.ANY, myhandler);
        btn2.addEventHandler(InputEvent.ANY, myhandler);

        Scene scene = new Scene(root, 300, 250);
        root.getChildren().addAll(btn1,btn2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

List-type nodes

Nodes like ComboBox, ListView, ChoiceBox have different events and properties,

e.g., a list of items (`ObservableList<String>`)
a way to format cells (`CellFactory`)
a way to read selected items (`getSelectionModel().getSelectedItem()`), …

The items can be “observed" for events

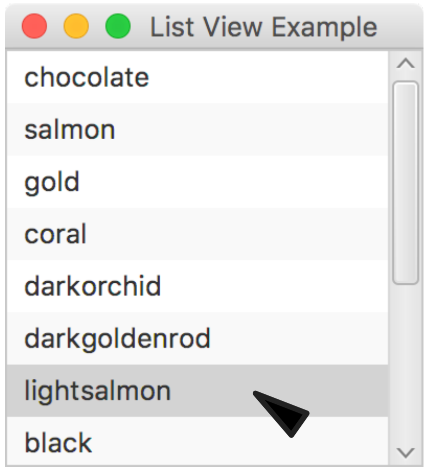
for each widget you need to check documentation + possible examples to get inspired (but **DO NOT PLAGIARIZE**)

List nodes (JavaFX)

```
public class ReactiveDropDownList extends Application {

    ListView<String> list = new ListView<String>();
    ObservableList<String> data = FXCollections.observableArrayList(
            "chocolate", "salmon", "gold", "coral", "darkorchid",
            "darkgoldenrod", "lightsalmon", "black", "rosybrown", "blue",
            "blueviolet", "brown");

    @Override
    public void start(Stage stage) {
        VBox box = new VBox();
        box.getChildren().addAll(list);

        list.setItems(data);

        list.setOnMouseClicked(event -> {
          String sel = list.getSelectionModel().getSelectedItem();
          System.out.println("New selection is " + sel);
        });

        Scene scene = new Scene(box, 200, 200);
        stage.setScene(scene);
        stage.setTitle("List View Example");
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

List items, that can be "observable" (advanced)

Returns selected item



(*) see online code on how to make cells colored

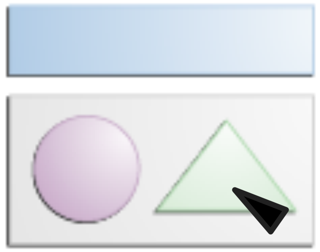
simple examples

In the online code you can find more examples:

- `OpeningWindow`: one window opens another
- `DraggingCursor`: changes cursor while mouse is pressed
- `MoveLabel`: changes the position of a label based on keyboard input (arrows)

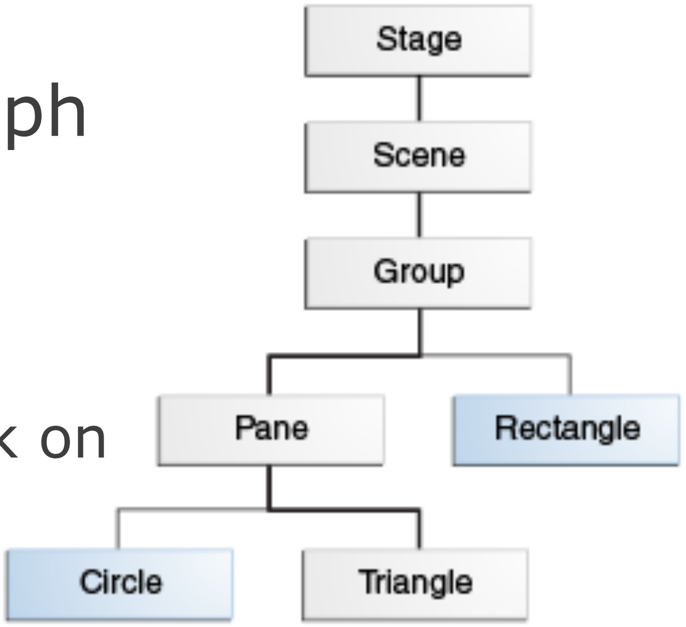
… have a look so we can discuss them next time

event chain



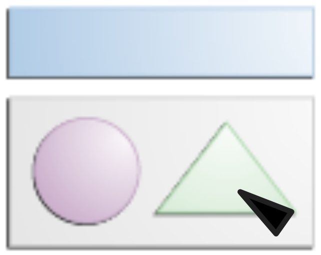
Imagine a scene like this one:

Which will result in a scene graph like this one:



… what happens when I click on the triangle?

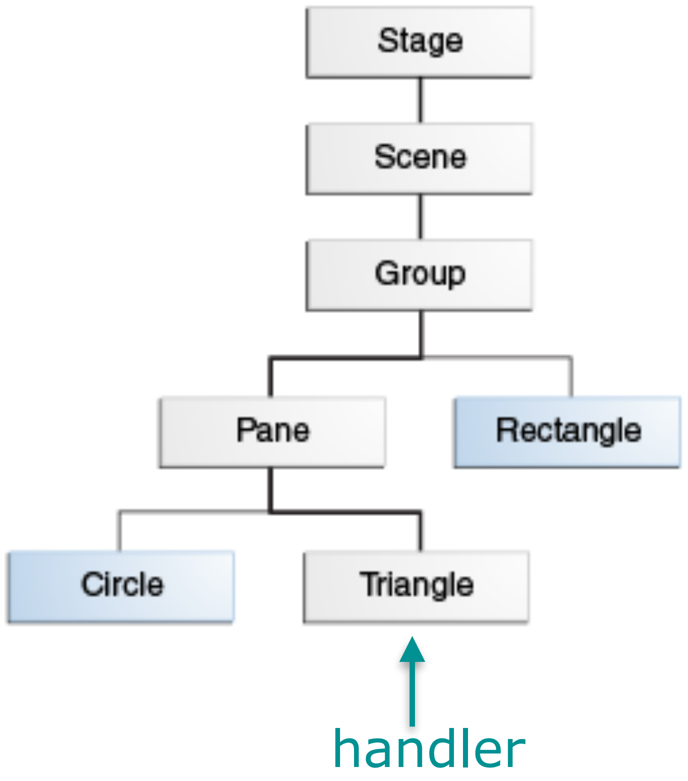
event chain (dispatching and bubbling)



1. event dispatching, goes downward through the hierarchy / dispatching chain

2. until a node can "handle" it, provide `handle():` target node

3. event bubbling, goes upwards the hierarchy and looks for other handlers

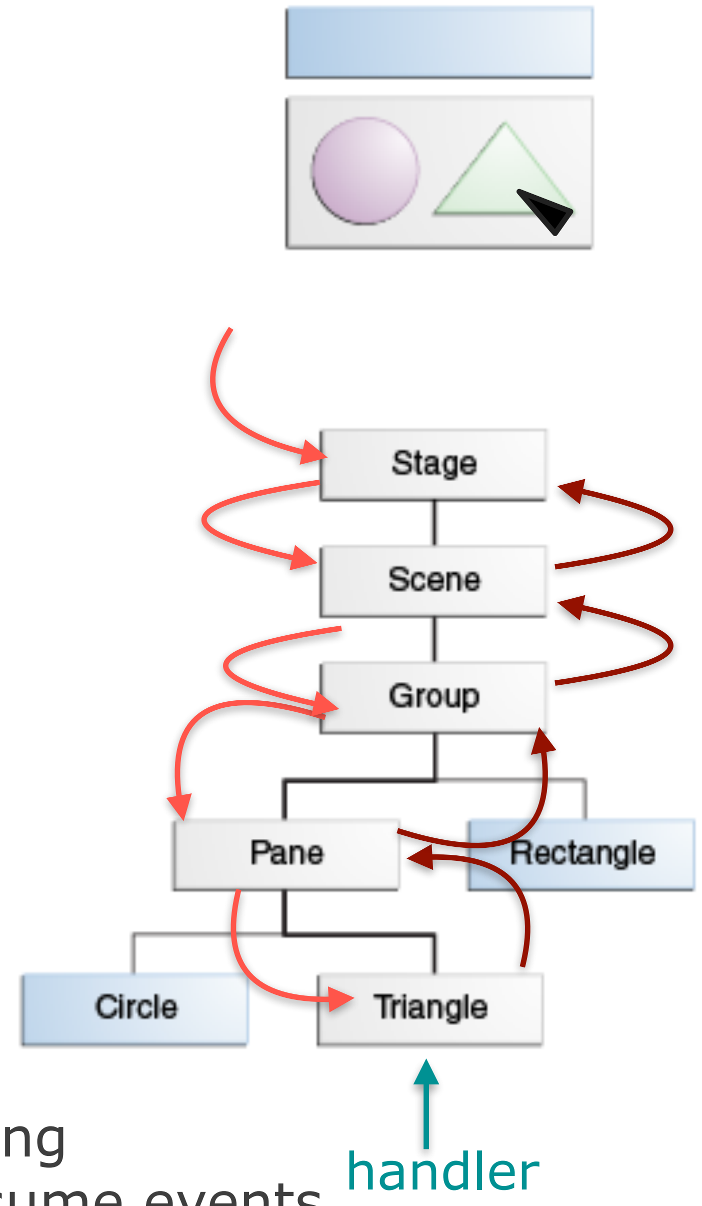


event chain (dispatching and bubbling)

1. event dispatching, goes downward through the hierarchy / dispatching chain (*)

2. until a node can "handle" it, provide `handle()`: target node

3. event bubbling, goes upwards the hierarchy and looks for other handlers



(*) EventFilters can be triggered in dispatching
EventFilters and EventHandlers can consume events

events have

Type	Event that occurred.
Target	The node that will handle the event (lowest in the hierarchy). For Key events the node that has focus, for mouse the node at mouse location
Source*	Origin of the event, with respect to the location of the event in the event dispatch chain. The source changes as the event is passed along the chain.

(*) We often use `getSource()` because when a handler is called we are in the target node (that has the handler)

event-driven programming

UI tookits support event-driven programming

We will talk about other event-handling
approaches (in other toolkits)
later in class

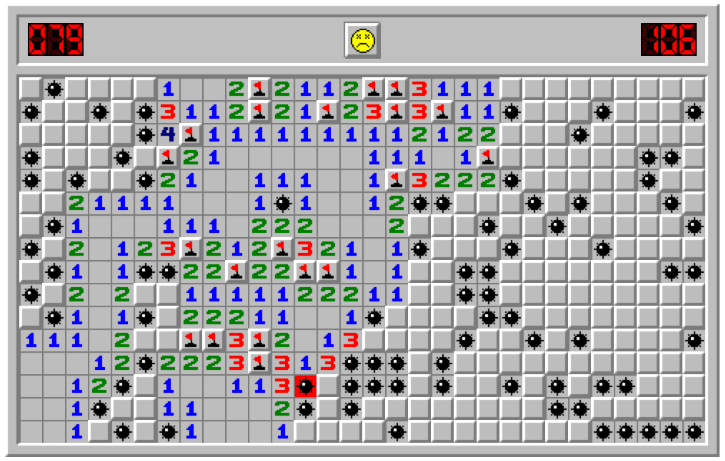
To be cont'd

Homework-2

Minesweeper v1: due on **Sep 27th (23:59)**

see announcement for details:

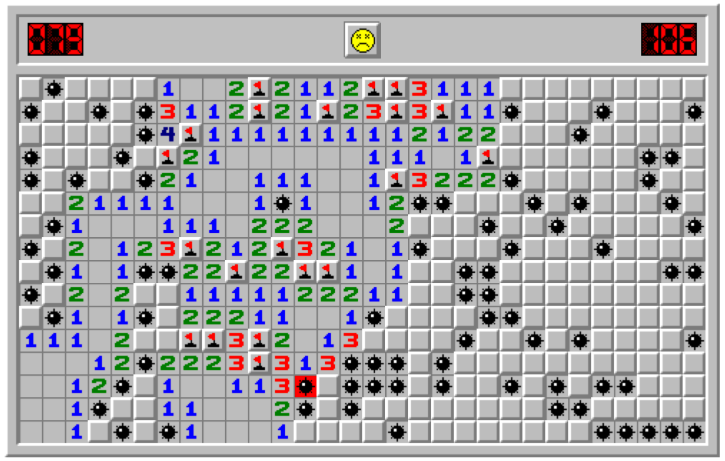
- code the layout of your windows
- code your UI elements
(put buttons and labels in cells for now)
- add one reaction to an input
(e.g., remove a button to
show a label)



In-class assignment

Consider your minesweeper,
how would you add functionality to remove* buttons when clicked?

* Consider the following possibilities
YourLayout.getChildren().remove(…)
YourButton.hide();



Resources

https://docs.oracle.com/javafx/2/events/jfxpub-events.htm