JavaFX and Event Handling

CSC02A2

Outline

Outline

Outline

Basic JavaFX
How JavaFX works
Scene Graph
Nodes
Basic JavaFX Application

Properties in JavaFX
Properties

GUI Event Handling
Events and Event Sources
Event Listeners
Inner Classes
Anonymous Class Listeners
Listener Interface Adapters

Events in code
Clickable Button
MenuBar with Menus and MenuItems
Lambdas

Hello World - JavaFX Style

Basic JavaFX

How JavaFX works I

GUIs (Grapical User Interfaces) provide a graphical way to interact with computer programs. JavaFX has been introduced as a replacement and improvement for older outdated approaches such as AWT and Swing.

JavaFX is designed to:

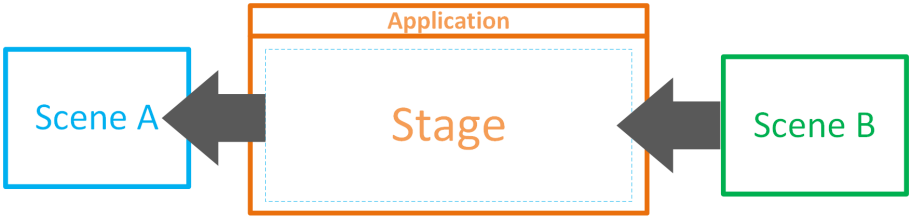
- Build rich Internet applications
- Run consistently across multiple platforms (desktop, web, mobile, etc.)

JavaFX uses the metaphor of a Stage to base its operation on:

Much like in the theatre, there is only one stage that is shown to the audience. As the show commences the props on stage and the actors change from scene to scene. In JavaFX, one scene can be displayed on the stage at a given time, but the scene can be swapped with another.

How JavaFX works II

Overview of JavaFX Scenes:



How JavaFX works III

JavaFX's scene-based approach aims to provide a simple and easy-to-change approach to delivering rich GUI applications.

`Scenes` enable `controls` for a particualr purpose to be contained within a `scene` to be loaded onto the `stage` when ready to be run, and taken off when you want to do something else.

Stage

A Stage is a container to manage one scene at a time.

Scene

A Scene is a container for a Scene Graph.

The Scene Graph I

If a `stage` can load a single `scene` at a time and swap it out with another, then what exactly is a `scene`?

In this metaphor the props and actors are realised in `JavaFX` as all of the controls and display components that need to be shown and interacted with.

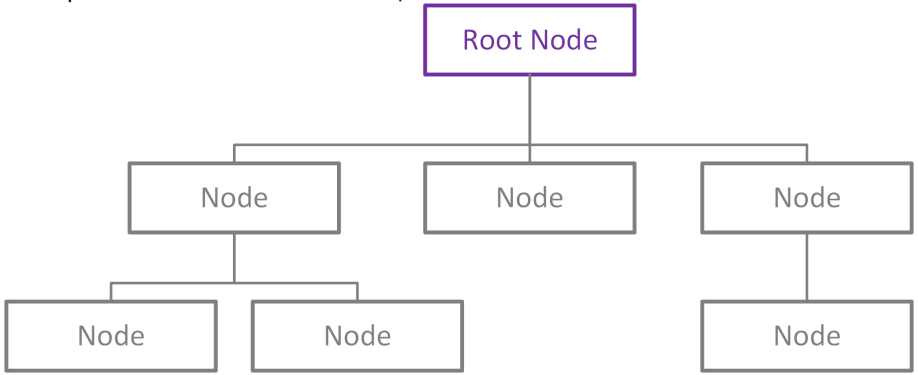
All of these are realised in the form of `Nodes` and are composed within the `Scene` in the form of the `Scene Graph`.

Scene Graph

A Scene Graph is a set of ordered nodes that descend from a root node to form a scene.

The Scene Graph II

Example of a node-based `Scene Graph`:



Nodes I

Everything in `JavaFX` is a `Node`.

Node

A node in the abstract superclass of all the graphical elements that a scene graph is made up of.

Examples of Nodes:

- `Button`
- `CheckBox`
- `Label`
- `MenuBar`
- `TextField`
- …and many more in `javafx.scene`

Nodes II

How do we manage the layout of nodes in the GUI?

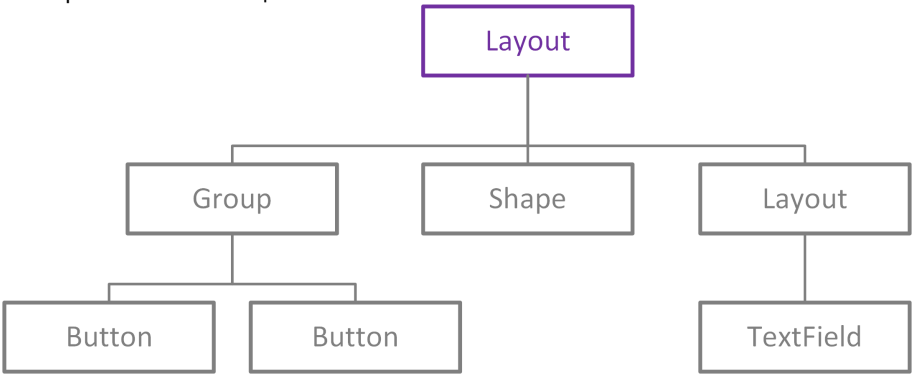
We can use different `nodes` from `javafx.scene.Layout` to do the layout for us. For example:

- `StackPane`
- `HBox`
- `VBox`
- `TilePane`
- `GridPane`
- `FlowPane`
- `AnchorPane`
- ...and many more

Each will position its child `nodes` differently based on the purpose of the `layout node`.

Nodes III

Example of a `Scene Graph`:



The JavaFX Application I

Structure of a JavaFX Application:

```
1  //Our program must extend Application
2  public class Main extends Application{
3    public static void main(String[] args) {
4      //We launch the Application by calling launch()
5      launch(args);
6    }
7    //start() is a method that must be overridden by any JavaFX
         Application
8    @Override
9    public void start(Stage primaryStage) throws Exception {
10     //start gives us a stage
11     //We need to create a scene (in this case just an anonymous
           scene) and load it onto the stage
12     primaryStage.setScene(new Scene());
13     //Open the curtains
14     primaryStage.show();
15   }
16 }
```


Properties in JavaFX

Properties I

JavaFX makes extensive use of special variables called properties.

Property

A property is a variable that can be observed (monitored for changes).

Properties allow for:

- Registering of listeners to watch any property and notify you when that property changes (more on this in a forthcoming lecture).
- Performing actions when properties change.
- Binding of one property to another so that when one changes another can automatically change as well.

Handling Properties I

Nearly all `JavaFX` UI elements are managed through `properties`.

For example: You can attach a `listener` to a colour `property` of a shape. If the colour changes, some code can be executed.

In general, this enables:

- Flexible event handling
- Responsive and consistent event handling

GUI Event Handling

Events and Event Sources I

Event

A signal that a certain *action* has taken place.

`Events` are used to send a signal to an application that something has occurred. Events may be triggered externally, such as user actions, or internally, specified by a programmer.

Within the context of `GUIs`, `nodes` signal whenever the user interacts with them in some way. These `nodes` are known as *source components*. An event object is created whenever an event occurs.

In `JavaFX` these come from `javafx.event.ActionEvent`

Event Listeners

Java uses a delegation based model for event handling. A `source object` *fires* an `event` and a `listener object` interested in handling the event will then respond.

The following conditions are required for an object to be a listener:

- The object must realise the appropriate event-listener interface.
- The object must be registered with the source object.

The object needs to realise the correct interface in order to have the method required to respond to the event.

In `JavaFX` these come from `javafx.event.EventHandler`

Inner Classes

`Classes` which are declared *inside* of another class are known as `inner classes`. Inner classes can access the instance variables/methods of the outer class.

Any visibility modifier can be used on the inner class. If the inner class is declared as static then the inner class can only access the static variables/methods of the outer class.

Creating instances of an inner class:

- Non-static inner class created outside outer class `OuterClass.InnerClass`
```
innerObject = outerObject.new InnerClass();
```
- Static inner class created outside outer class `OuterClass.InnerClass`
```
innerObject = new OuterClass.InnerClass();
```


Anonymous Class Listeners

An `anonymous inner class` is an inner class that does not have a name and combined declaring and instantiating the class into a single step. The following conditions apply to anonymous classes:

- Anonymous classes always extend/implement a class/interface but do not use the keywords.
- Anonymous classes cannot be abstract.
- Anonymous classes always uses the zero argument constructor if present.

Listener Interface Adapters

Since `interfaces` require all `methods` to be implemented, classes which implement listener interfaces *need to implement all the event methods* even if they will not respond to those events.

`Listener adapter classes` are classes that implement the listener interfaces and provide an empty default implementation. Only methods that need to be handled are overridden. This reduces the amount of code that is required to be written.

Time for examples:

Events in code

Clickable Button

Create a button and register and event listener:

```
1  Button btn = new Button();
2  btn.setText("Say 'Hello World'");
3  //Create anonymous EventHandler inner class for event handling and register
   ↪  it to the OnAction event listener of the button
4  btn.setOnAction(new EventHandler<ActionEvent >() {
5    //Implement the required method to handle event
6    @Override
7    public void handle(ActionEvent event) {
8      //Do something
9    }
10 });
```


MenuBar with Menus and MenuItems

Create a menu bar, a menu, and menu item, then connect them:

```
1  MenuBar menuBar = new MenuBar();
2  Menu menu = new Menu("Menu Text");
3  //Add menu to menu bar
4  menuBar.getMenus().add(menu);
5  MenuItem mi1 = new MenuItem("Menu Item Text");
6  //Add menu item to menu
7  menu.getItems().add(mi1);
8  //Add action listener to menu item
9  mi1.setOnAction(new EventHandler<ActionEvent >() {
10   //Implement the required method to handle event
11   @Override
12   public void handle(ActionEvent event) {
13     //Do something
14   }
15 });
```


Lambdas (for the brave) I

In Java 8, lambda expressions are Java's first step into `functional programming` (covered in the Honours course of the same name).

Lambda

An expression which is a function that can be created without belonging to any class.
Can be passed around as if it was an object and executed on demand.

Lambda Format

```
parameters -> statements to be executed
```

For more information on Lambdas and their usage see:
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

This is useful for us because it saves us having to instantiate `Inner Classes`!

Lambdas (for the brave) II

Anonymous inner class vs a lambda:

```
1  btn.setOnAction(new EventHandler<ActionEvent >() {
2    //Implement the required method to handle event
3    @Override
4    public void handle(ActionEvent event) {
5      //Do something
6    }
7  });
8
9  /* Becomes: */
10
11 btn.setOnAction(e -> {
12   //Do Something
13 });
```


Hello World - JavaFX Style

JavaFX Hello World I

HelloWorld main:

```
1  //Must extend Application (JavaFX)
2  public class Main extends Application{
3
4    public static void main(String[] args) {
5      launch(args); //We launch the application from here
6    }
7    //Class scope attribute to store text
8    private TextField txt = new TextField();
9
10   //Start method from Application (JavaFX entry point)
11   @Override
12   public void start(Stage primaryStage) throws Exception {
13     //start() gives us our stage from Application
14     primaryStage.setTitle("Hello World!");
15     Button btn = new Button();
16     btn.setText("Say 'Hello World'");
17     btn.setOnAction(new EventHandler<ActionEvent >() {
18
19
```


JavaFX Hello World II

```
20        @Override
21        public void handle(ActionEvent event) {
22          txt.setText("Hello World!");
23        }
24      });
25
26      //Create root node
27      VBox root = new VBox();
28      root.getChildren().add(btn);
29      root.getChildren().add(txt);
30
31      //Set the scene for the stage
32      primaryStage.setScene(new Scene(root, 300, 250));
33
34      //"Open the curtains"
35      primaryStage.show();
36    }
37  }
```


JavaFX Hello World III

Don't forget the imports:

```
1  //Application provides us with an entry point for JavaFX
2  import javafx.application.Application;
3  //Event allows us to interact with events and properties
4  import javafx.event.ActionEvent;
5  import javafx.event.EventHandler;
6  //Scene gives us controls for use in a scene
7  import javafx.scene.Scene;
8  //Controls allow for interaction with the user
9  import javafx.scene.control.Button;
10 import javafx.scene.control.TextField;
11 //Layouts allow for nodes to be arranged automatically
12 import javafx.scene.layout.VBox;
13 //Stage gives us a stage to load scenes
14 import javafx.stage.Stage;
```