SOFT3202

Adam Ghanem

March 30, 2022

Contents

1 Tutorial 5 EXPORT

1.1 GUIs

1.1.1 Event Driven Programming

Most programmers have learnt how to develop in the form of procedural code. The code is responding to actions from the user. This enforces the idea that the code has some sort of flow of execution, and makes it necessary to start to look at concurrent solutions. This flow is dependent upon that actions of the user. **Development is not defined as performing actions in some sort of procedure but rather as responding to events that are occurring.**

This level of programming depends on a continuous loop that is always on the

alert for upcoming events. The goal is to separate event logic from the rest of the program logic. By using an event driven approach we must **detach the interaction of GUI elements such that order is no longer related.**

1.1.2 JavaFX

NOTE: You do not need to use JavaFX (it is recommended). You can make use of `awt` or `swing`. Very popular to learn some Web and Android studio level environments to create some applications. JavaFX development can seem a bit slow, but it has a decent amount of features that is useful to get a basis of understanding GUI programming. If you think it looks ugly you can be customise it using `CSS`, but I really don't think that would be enjoyable as well.

`https://docs.oracle.com/javase/8/javafx/get-started-tutorial/img/css-style-sample.gif`

Recall we have to add a `JavaFX` dependency into our `build.gradle` file:

```
plugins {
    id 'application'
    id 'org.openjfx.javafxplugin' version '0.0.10'
}

sourceCompatibility = 1.17
targetCompatibility = 1.17

repositories {
    mavenCentral()
}

javafx {
    version = "17.0.2"
    // We are including only javafx controls
    // Look here if you are getting import errors
    modules = [ 'javafx.controls' ]
}

dependencies {
```

```
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'
    testImplementation group: 'org.mockito', name: 'mockito-junit-jupiter', versio
    testImplementation group: 'org.mockito', name: 'mockito-core', version: '2.19.
    testImplementation 'org.hamcrest:hamcrest:2.2'
    implementation 'com.google.code.gson:gson:2.7'
}

application {
    mainClass = 't5.App'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

Please go through these readings to go through how JavaFX works:

- https://docs.oracle.com/javase/8/javase-clienttechnologies.htm
- https://docs.oracle.com/javase/8/javafx/scene-graph-tutorial/scenegraph.htm#JFXSG107
- https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm#JFXST788
- https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm
- https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm#JFXUI336
- https://openjfx.io/index.html
- https://fxdocs.github.io/docs/html5/

I will not go through these in detail just sections which will be useful for initial GUI development.

HelloWorld the FX Style

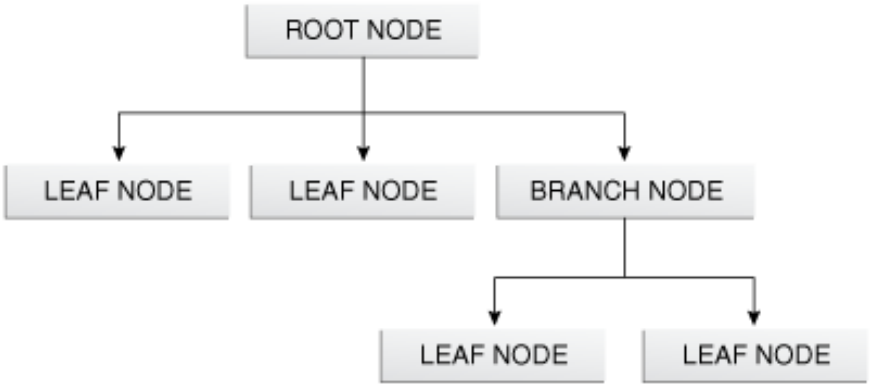
```
package t5;
```

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {

  @Override
  public void start(Stage primaryStage) {
    // Create an interactable button
    Button btn = new Button();
    btn.setText("Say 'Hello World'");
    // This is what we refer to as when we have event driven programming
    btn.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
          System.out.println("Hello World!");
        }
      });

    // Just like our border pane, this sets the boundaries for our buttons etc.
    StackPane root = new StackPane();
    root.getChildren().add(btn);

    // Scenes take in a pane/root and the stage renders the scene
    Scene scene = new Scene(root, 300, 250);
    primaryStage.setTitle("Hello World!");
    primaryStage.setScene(scene);
    primaryStage.show();
  }

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

As we can see above we have that HelloWorld:

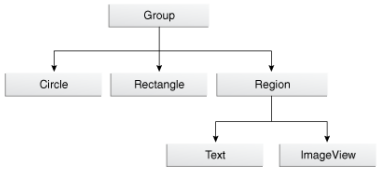
- `extends Application`, this is how we can call `launch(args)` (which it inherits)
- we go from `main` and then `start` will override
- unlike procedural programming we have no idea what `launch(args)` is doing, usually sets up the environment
  - we have some buttons that can be clicked on at anytime
  - within that start method part of it is procedural
  - however this scene will run on a loop

Components: From `https://docs.oracle.com/javase/8/javafx/scene-graph-tutorial/scenegraph.htm`:



Description of "Figure 1-1 Root, Branch, and Leaf Nodes"

Figure 1-2 Specific Root, Branch, and Leaf Classes



Description of "Figure 1-2 Specific Root, Branch, and Leaf Classes"

In Figure 1-2, a `Group` object acts as the root node. The `Circle` and `Rectangle` objects are leaf nodes, because they do not (and cannot) have children. The `Region` object (which defines an area of the screen with children than can be styled using CSS) is a branch node that contains two more leaf nodes (`Text` and `ImageView`). Scene graphs can become much larger than this, but the basic organization — that is, the way in which parent nodes contain child nodes — is a pattern that repeats in all applications.

Basic structure:

- We have a Stage, which is our window
- then this holds a Scene
- and from it we have a root/pane
- which we can see resembles a tree structure

We can have multiple components:

- from shapes
- to labels
- textfields
- buttons
- etc

Layout Management It can be a bit finnicky working with coordinates and trying to get them to fit. Instead, in JavaFX we have ways of managing the layout of our child objects. The most common is BorderPane, and there are more complex ones you can micro-manage yourself.

Border Pane So a border pane in JavaFX is basically used for the root of our scenes. It splits our children (objects within our pane) into 5 positions - Top, Left, Bottom, Center and Right. With this obviously comes the methods for `setCenter(Node value)`, etc.. and `getRight()` to set and get the nodes, respectively. `https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm`

Events For GUI programs we need to have events, all these events are essentially loaded from the start. We attach a listener to all of our events which trigger our events. Just like what you may have experienced from other courses, it can listen to mouse movements, acceleration, clicks, holding etc. These events are triggered and pass down from the root node which then propagates down to the target node and then back up. The reason we have this is because we can have events handled in different orders, they can be consumed or have different responses.

A snippet from the example above:

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
      System.out.println("Hello World!");
    }
  });
```

These resemble our anonymous classes, as a form of our functional interface. One of the more modern ways to do this is to make use of a `lambda`:

```
btn.setOnAction(event -> {
      System.out.println("Hello World!");
    });
```

Model-View Separation This is one of the most important design principles regarding GUI programming. However, unlike what we have in procedural programming we should:

- Try not to create utility methods to help with reuse, it will complicate things down the line

But in line with what we learn last week:

- Composition »» Inheritance
- We do not wnat to devolve back down to `instanceof` calls when we get parent objects.

Model:

- Essentially it is the high level representation of what our application is
- It must be completely separate from the `View` at all times, remember the SRP principle
- we have the application specific code here e.g. logic, data, algorithms
- it is our representation of content

View:

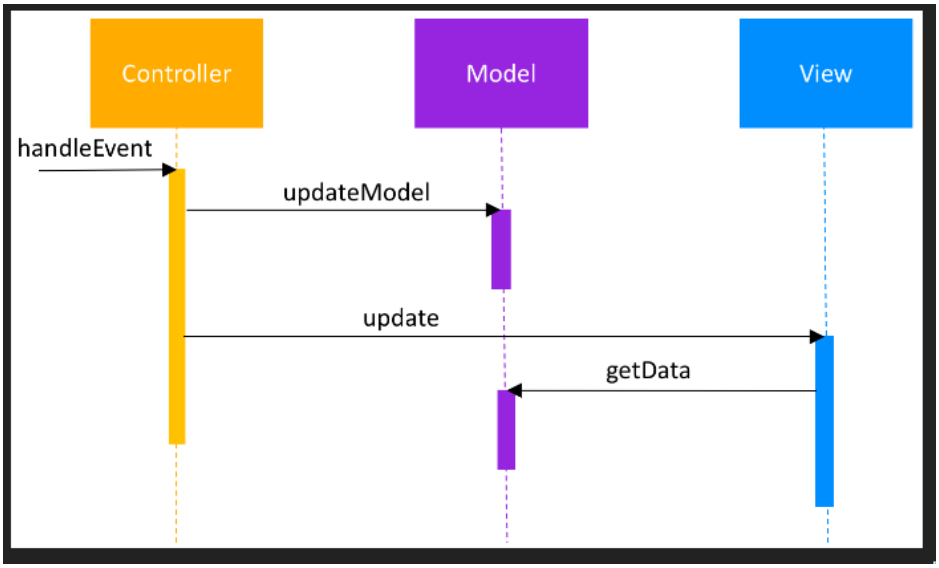
- this is our structure and layout of what we see on the GUI

- **it is the representation of the model** and has to be coupled to it
  - views can be swapped out for others
- this is where we receive the user interactions through keyboards etc.
- it can forward these requests to a controller

Controller (optional to leave out in some simpler scenarios):

- takes in input and delegates it to either the Model or the View

From `https://upday.github.io/blog/model-view-controller/`:



But how do we ensure this we need to ensure we highlight areas where:

- there is coupling between view and logic
- imports
- creating objects
- calling methods

How does this separation work:

- the view in this case will send the intruction to the model

  - like an observer it can update itself after every instruction and be notified when the model changes.
- the model will react to this change

We can split this in multiple ways:

- either have a single point of contact talking to the model
- or multiple views talking to sections of the model

1.2 HTTP

The Hypertext Transfer Protocol (HTTP) is essentially the most popular data communication tool and acts as the foundation for the Web. Mostly all software interacts or uses this. In this task it will not be expanded upon in too much details. When a server is run with HTTP pages then event driven programming comes in to deal with the asynchronous nature of its use.

- there is a client making requests (GET or POST) to a server
- GET is for retrieving information
- POST is for sending information

The process is:

- build request
- send request
- GET
- interpret response

1.2.1 Request

A request will have:

- a domain
- target `URI`
- resource
- headers

- in this we can have credentials or pass specific queries through:
  - for example take a look at this we have the URL and then details being sent through `v=...`
  - `https://www.youtube.com/watch?v=dQw4w9WgXcQ`
- can also have a body in POST, e.g. send content through it using JSON.

In this unit, the synchronous form will be dealt with.

- we use it for sending and receiving data
- doesn't just have to be hypertext
- **in most cases you will be using it for JSON**, asking for JSON from webservers
- there are alternatives GSON and you can use simple JSON if you are familiar with it.

1.2.2 Response

A response is retrieved after you make a POST to a server or API:

- contains header and body
- has a status code
- note [] = inclusive, () = exclusive

  There are multiple codes, you can think of these as status codes.

Code	Category
[100, 200)	Informational
[200, 300)	Success
[300, 400)	Redirection
[400, 500)	Client Error
[500, 600)	Server Error

Java provides an inbuilt HTTP client which you can access with: `java.net.http.HttpClient`. You can test on a server `https://jsonplaceholder.typicode.com`. Provides access points, is immutable but provides fake data.

1.2.3 Tasks

The example from the Lecture:

```
import com.google.gson.Gson;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class HelloHTTP {
  public static void postRequest() {
    try {
      // We are storing the body here, can use different formats other than a simp
      Post post = new Post(10, "My title", "My body text\nOf this post");
      Gson gson = new Gson();
      String postJSON = gson.toJson(post);

      // Create the HTTP request, but in this case are adding a new post
      HttpRequest request = HttpRequest.newBuilder(new URI("https://jsonplacholde
        .POST(HttpRequest.BodyPublishers.ofString(postJSON))
        .header("Content-Type", "application/json")
        .build();

      // We get the result of adding our previous post
      HttpClient client = HttpClient.newBuilder().build();
      HttpResponse<String> response = client.send(request, HttpResponse.BodyHandle

      // The result of adding the post e.g. status codes
      System.out.println("Response status code was: " + response.statusCode());
      System.out.println("Response headers were: " + response.headers());
      System.out.println("Response body was:\n" + response.body());
    } catch (IOException | InterruptedException e) {
      System.out.println("Something went wrong with our request!");
      System.out.println(e.getMessage());
    } catch (URISyntaxException ignored) {
```

```
    // This would mean our URI is incorrect - this is here because often the URI
    // hard-coded and so needs a way to be checked for correctness at runtime.
      }
}

public static void getRequest() {
  try {

    // BUILDER to get a URI (in this case the dummy one we will be testing on) a
    // A typo results in a URI Syntax Exception
    HttpRequest request = HttpRequest.newBuilder(new URI("https://jsonplaceholde
      .GET()
      .header("Content-Type", "application/json")
      .build();
    HttpClient client = HttpClient.newBuilder().build();

    // This here is the asynchronous section of our code, recall await and asyn
    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandle

    // Messages based on status code
    System.out.println("Response status code was: " + response.statusCode());
    System.out.println("Response headers were: " + response.headers());
    System.out.println("Response body was:\n" + response.body());

    // Now creating our POST object
    Gson gson = new Gson();
    Post post = gson.fromJson(response.body(), Post.class);
    System.out.println(post);
  } catch (IOException | InterruptedException e) {
    System.out.println("Something went wrong with our request!");
    System.out.println(e.getMessage());
  } catch (URISyntaxException ignored) {
    // This would mean our URI is incorrect - this is here because often the URI
    // hard-coded and so needs a way to be checked for correctness at runtime.
  }
}
public static void main(String[] args) {
  getRequest();
```

```
    postRequest();
  }
}
```

The post class is not a REST Post but a blog post, the name here is deceiving.

```
public class Post {
  private int userId;
  private String title;
  private String body;

  public Post(int userId, String title, String body) {
    this.userId = userId;
    this.title = title;
    this.body = body;
  }
  public int getUserId() {
    return userId;
  }
  public String getTitle() {
    return title;
  }
  public String getBody() {
    return body;
  }
  @Override
  public String toString() {
    return "User ID: " + userId + " | " + title + "\n" + body;
  }
}
```

- for fun see the 404 game