Lab 10: JavaFX

Author: Yida Tao, Yao Zhao

Background

If you are using Java 8, JavaFX is already bundled with your JDK so you do not need to take any further steps. With the advent of Java 11 in 2018, however, JavaFX was removed from the JDK so that it could evolve at its own pace as an independent open-source project guided by Oracle and others in the OpenJFX community.

In this tutorial, we assume that you're using Java 11 or later.

Create a "Hello World" JavaFX Application

Prerequisite

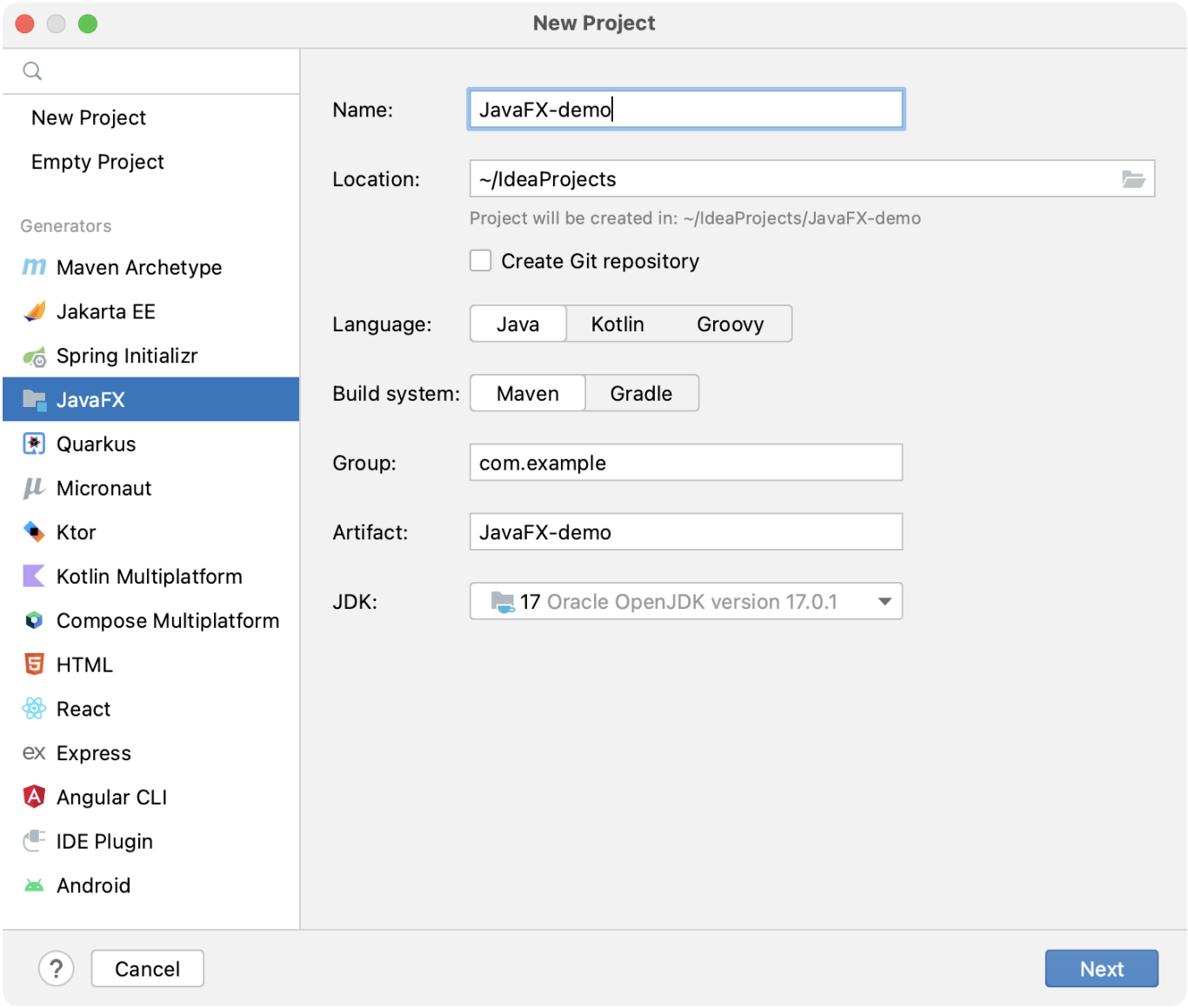
To be able to work with JavaFX in IntelliJ IDEA, the JavaFX bundled plugin must be enabled:

- In the `Settings/Preferences` dialog (Ctrl+Alt+S), select `Plugins`.
- Switch to the `Installed` tab and make sure that the JavaFX plugin is enabled. *If the plugin is disabled, select the checkbox next to it*.
- Apply the changes and close the dialog. Restart the IDE if prompted.

Create a new project

When you create a new JavaFX project, IntelliJ IDEA generates a fully configured sample application.

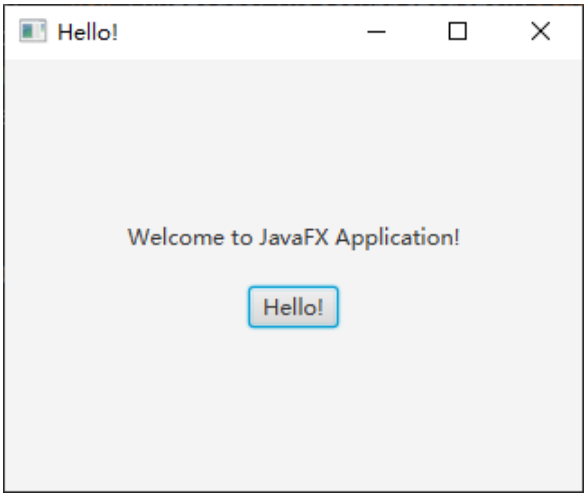
- Launch IntelliJ IDEA. Click `New Project`. Otherwise, from the main menu, select File | New | Project.
- From the Generators list on the left, select `JavaFX`.
- Name the new project, change its location if necessary, and select a language, and a build system.
- In the Group field, specify the name of the package that will be created together with the project.
- From the JDK list, select the JDK that you want to use in your project. If the JDK is installed on your computer, but not defined in the IDE, select Add JDK and specify the path to the JDK home directory. If you don't have the necessary JDK on your computer, select Download JDK.
- Click Next -> Create.



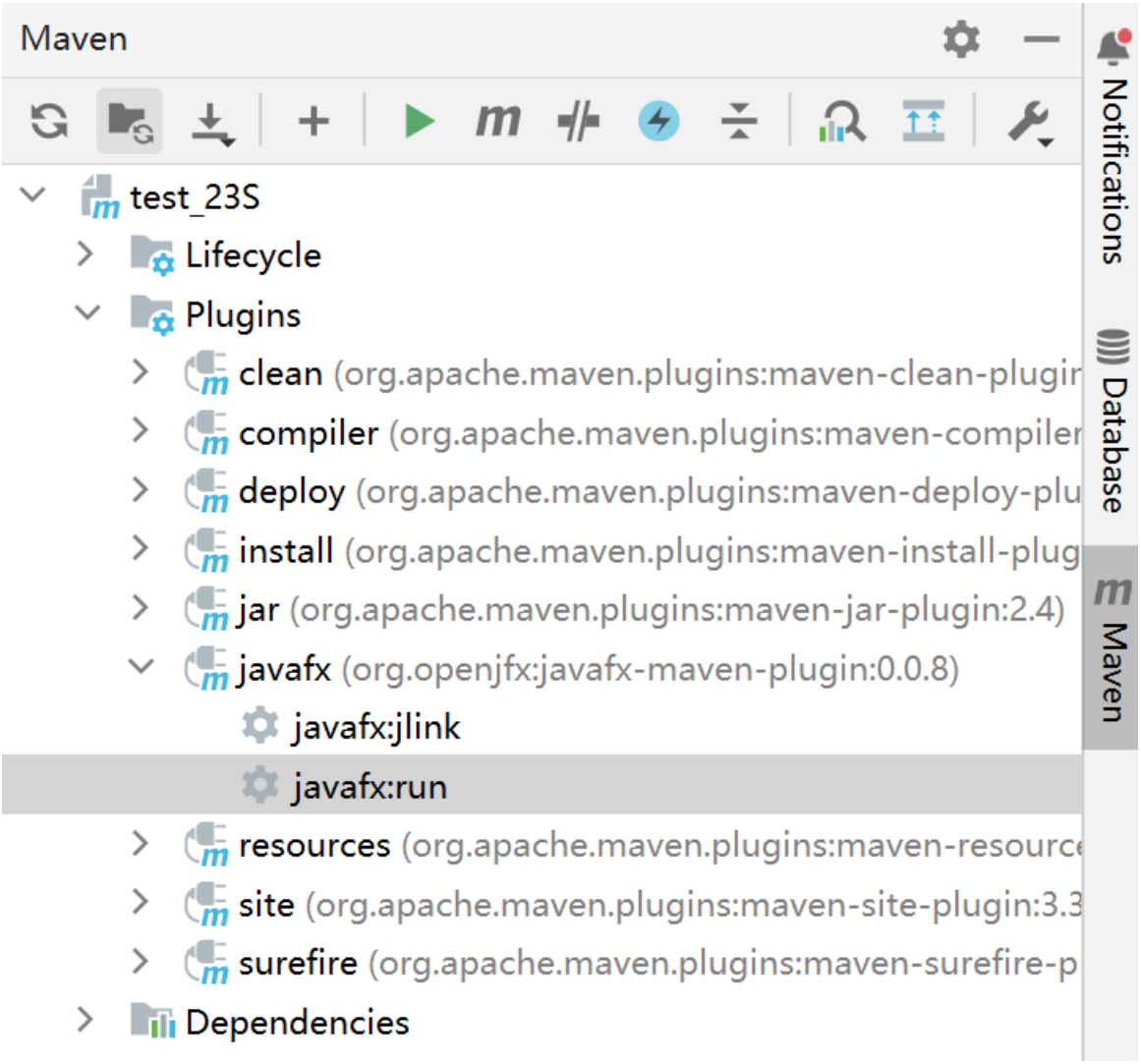
Run the application

Open the `HelloApplication.java` class, click the `Run application` icon in the gutter, and select `Run 'HelloApplication.main()'`. The IDE starts compiling your code.

When the compilation is complete, the application window appears. This means that the project is configured correctly and everything works as it should.



You may also open the "Maven" view, select the current project, then click "Plugins" -> "javafx" -> "javafx:run" to execute the program. Here, we are executing the javafx maven plugin's `run` goal, which is configured in `pom.xml` to execute the main class.



Add JavaFX Functionalities to an Existing Project

Suppose that you have an existing Java project, and you want to use JavaFX in it. In this case, you should add JavaFX support to your current project. You could achieve this using either of the following approaches.

Using Maven

You could let maven automatically download the required JavaFX dependencies by put your mouse on missing dependencies and click the suggested action.

Then, add a `module-info.java` to your project, which contains:

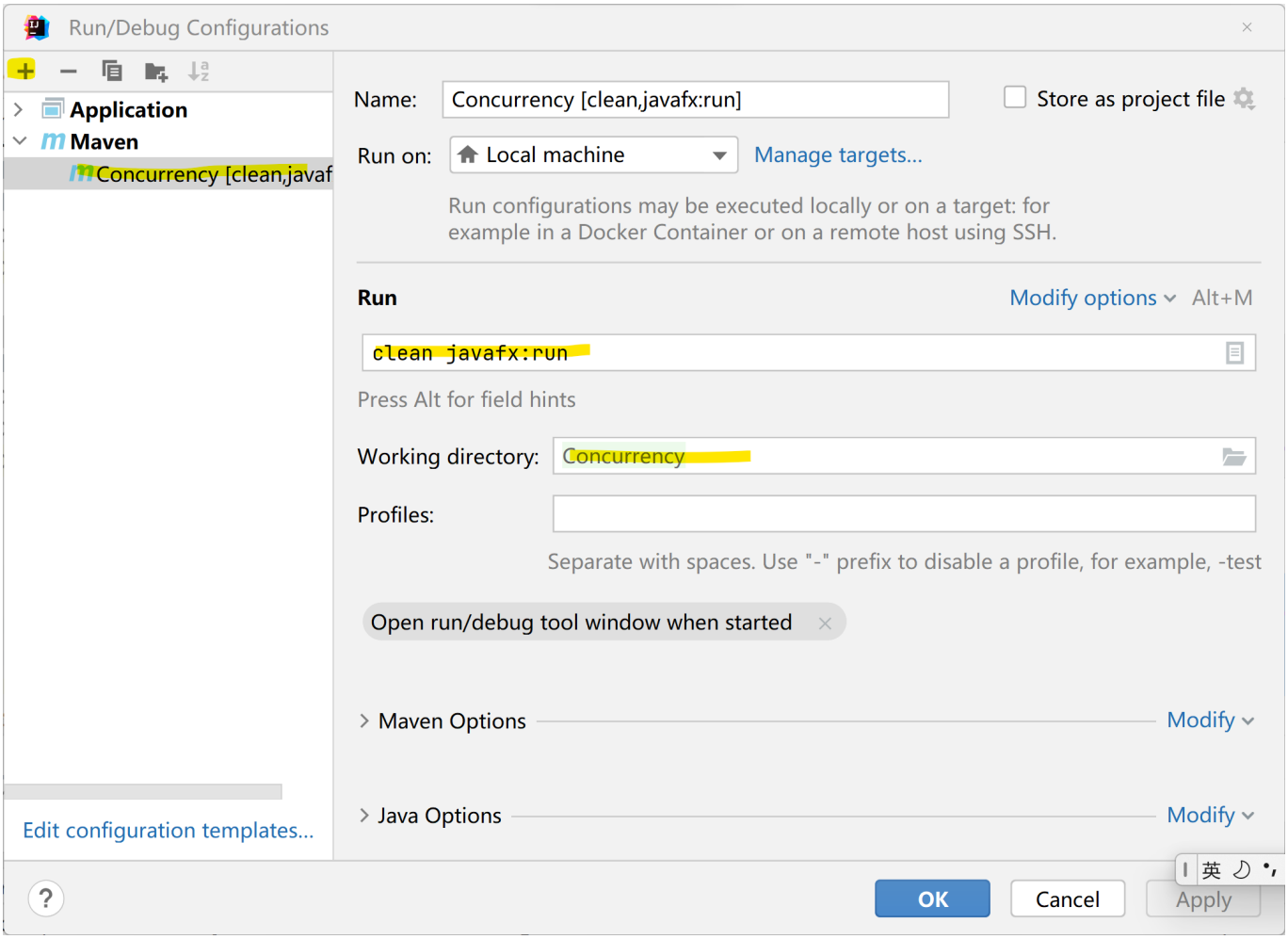
```
module Concurrency {
    requires javafx.controls;
    requires javafx.fxml;

    // . should be replace by your package name
    // . stands for default package
    opens . to javafx.fxml;
    exports .;
}
```

Next, add the following to `pom.xml`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <release>17</release>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-maven-plugin</artifactId>
            <version>0.0.8</version>
            <configuration>
                <mainClass>ConcurrencyExample</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Next, click `Edit Configuration`->`+`->`Maven`, in `Run`, input `clean javafx:run` for your working directory.

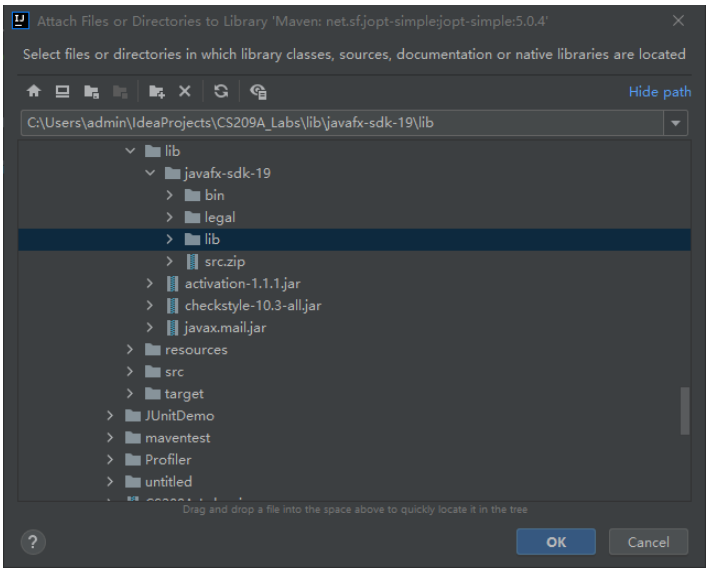


Click `Apply`. Now you can run this configured maven command to start your JavaFX application.

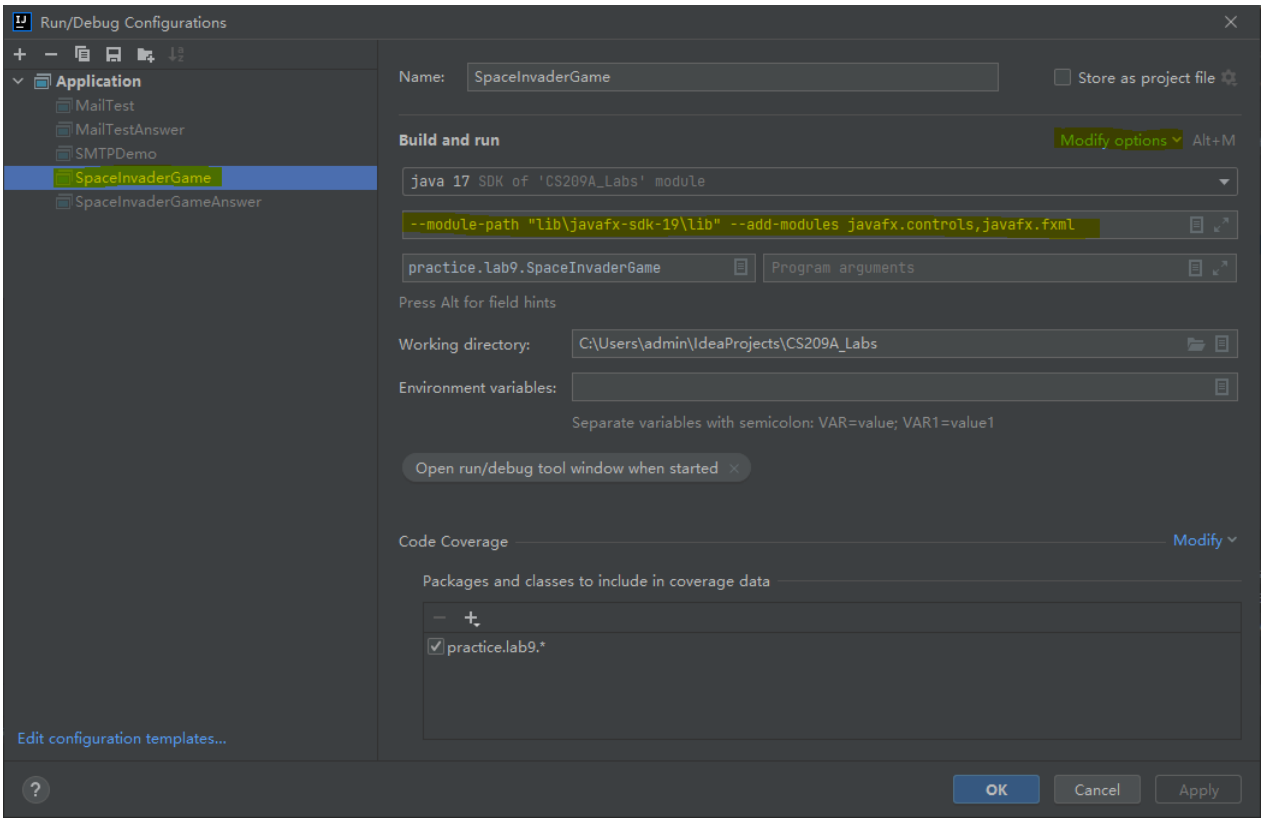
Manually

Alternatively, you could manually download JavaFX and configuring the classpath.

- Download JavaFX SDK from the official site. For example, download the Windows/x86/SDK and unzip the downloaded zip for Windows users.
- In the existing project, click `File` -> `Project Structures` -> `Project Settings` -> `Libraries`, click `+`, find your downloaded javafx-sdk folder and select the `lib` subfolder.



- Click `Run` -> `Edit Configurations`, select your JavaFX class on the left, click `Modify options` -> `Add VM options`, and add `--module-path "your-path-to-sdk\javafx-sdk-19\lib" --add-modules javafx.controls,javafx.fxml`

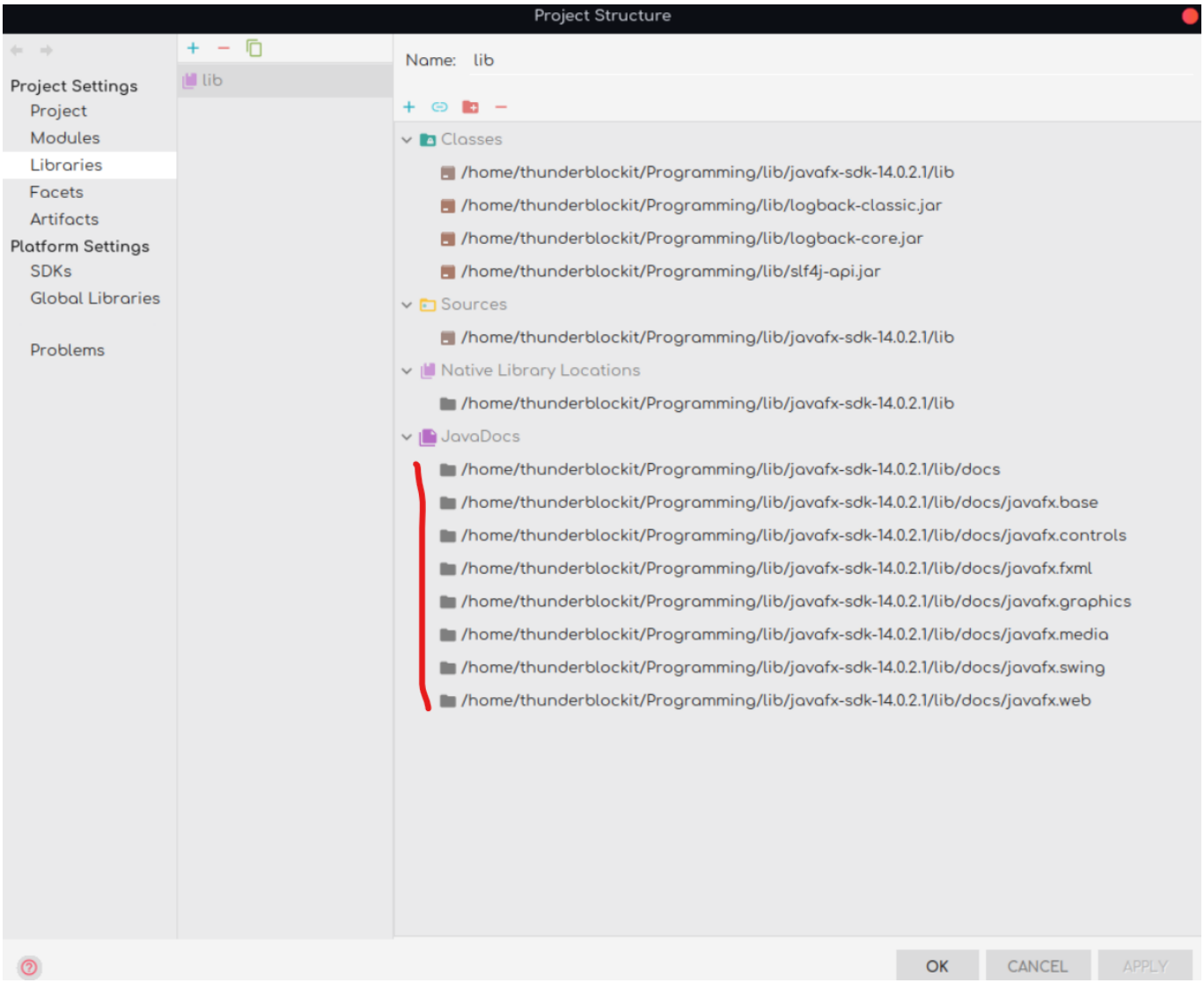


Now, you should be able to run your JavaFX code in this project.

Adding JavaDoc Support

Documentation is super helpful for us to learn a library. If you want to bring up JavaFX's documentation in IDEA, right click `pom.xml` in your project, select "maven" -> "download documentation", which could automatically download corresponding documentation for your javafx dependency.

Alternatively, if you don't have `pom.xml` in your project, you could also download the JavaFX's Javadoc here (be sure to use the same version as JavaFX SDK), unzip it, then add it as "JavaDocs" in `Project Structure` -> `Project Settings` -> `Libraries`.

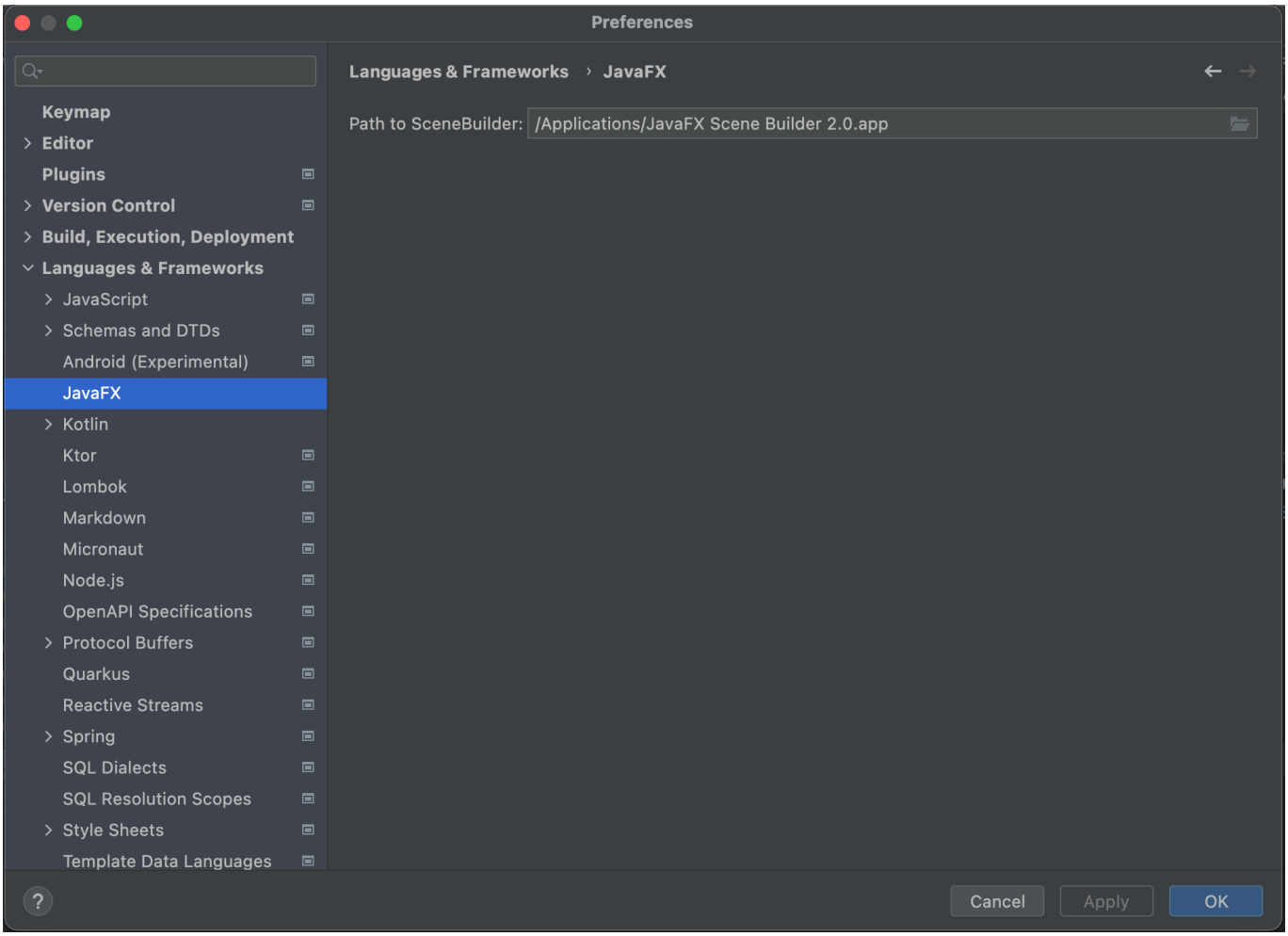


Once applying the above configuration, whenever you hover your mouse to a certain JavaFX entity in IDEA, you should be able to see its JavaDoc automatically pops up.

Using Scene Builder for UI Design

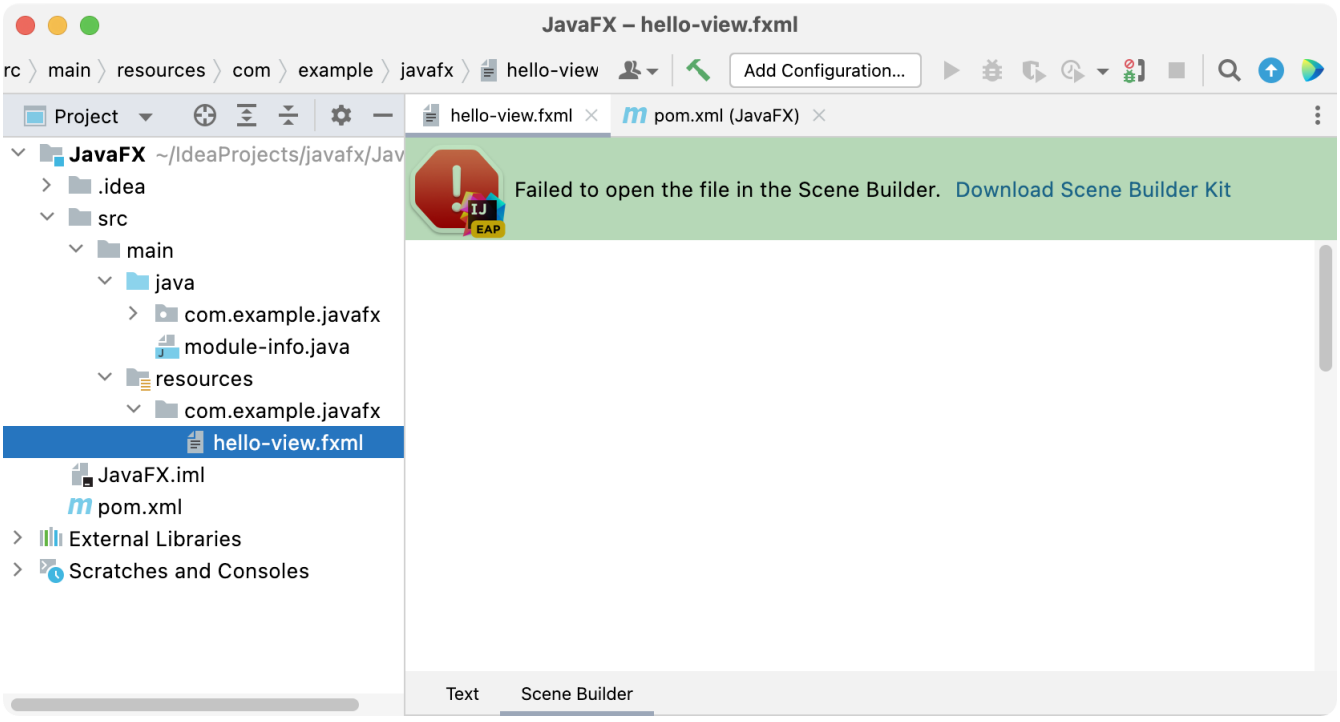
First, download Scene Builder here and install it.

Next, click "File" -> "Setting" -> "Language & Frameworks" -> "JavaFX", and specify the installation path for Scene Builder.

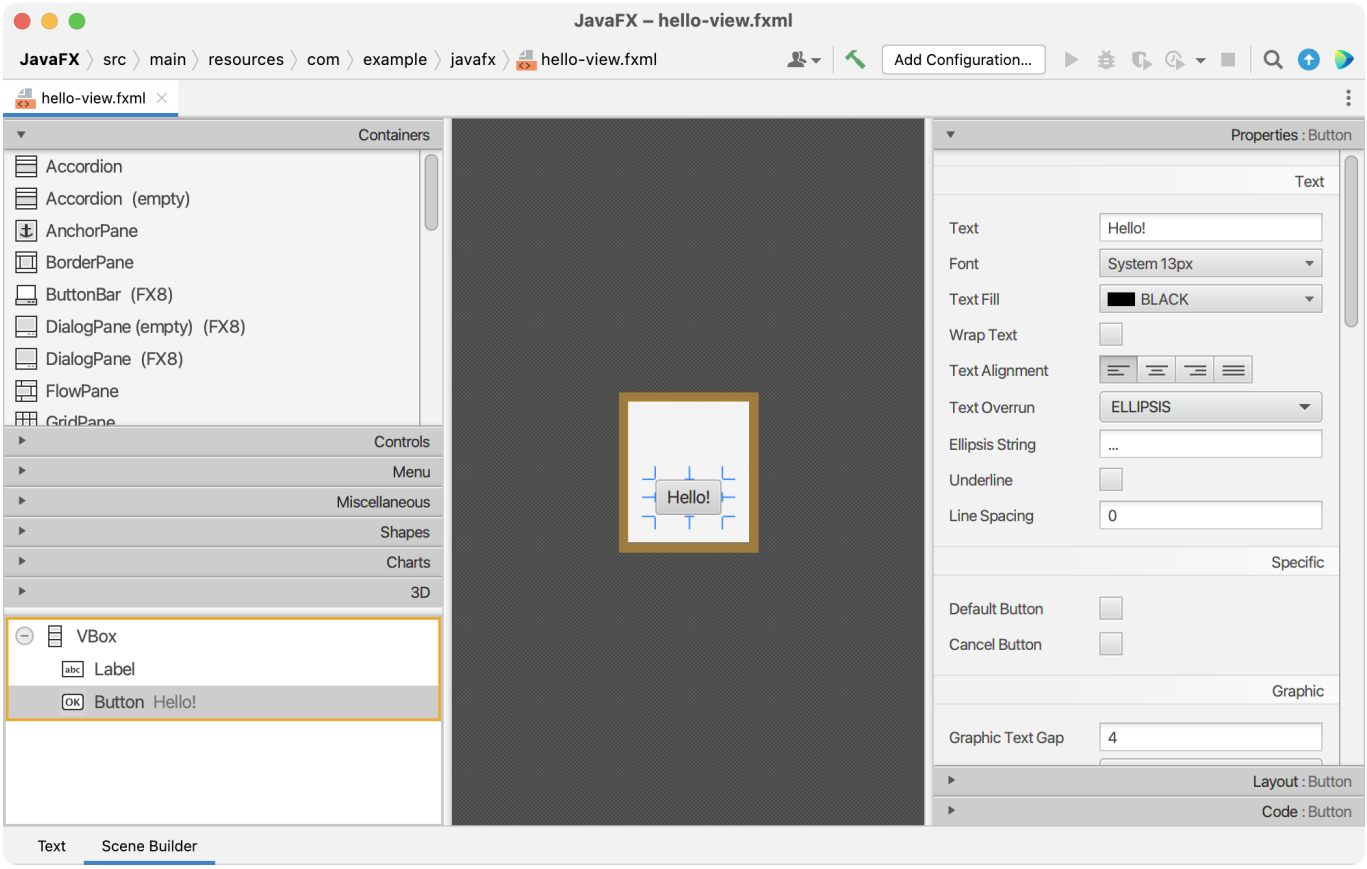


Now, when you open an `.fxml` file in the editor, there are two tabs underneath the editing area: the Text tab is for developing the markup, and the Scene Builder tab is for editing the file in Scene Builder.

If you cannot view `.fxml` in the Scene Builder tab, you'll see a notification like below. In that case, click `Download Scene Builder Kit` in the notification to download and install the tool.



Next, you could open the `.fxml` in the Scene Builder tab and design the UI visually.



If this still doesn't work, you could right-click .fxml and select "Open in Scene Builder", which will open the file in Scene Builder rather than inside IDEA.

JavaFX - Concurrency Example

Run `ConcurrencyExample.java`, observe the results. The reason why the window hangs is because the JavaFX scene graph, which represents the graphical user interface of a JavaFX application, is not thread-safe and can only be accessed and modified from the UI thread also known as the `JavaFX Application thread`. As we're implementing a long-running task on the JavaFX Application thread, the UI becomes unresponsive until the task is done.

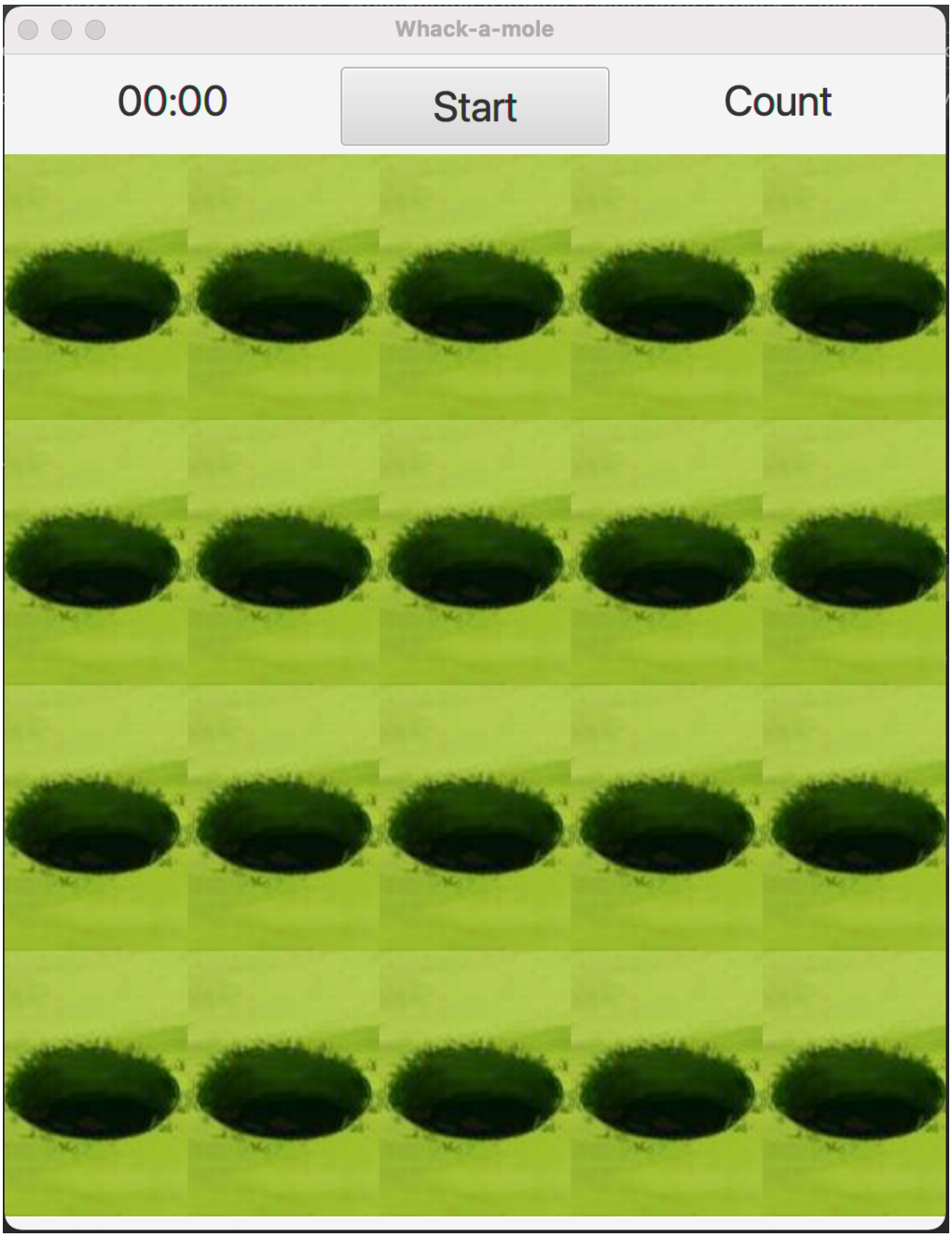
For this reason, we typically do long-running tasks on one or more **background threads** and inform the JavaFX Application thread to update the UI. `Platform.runLater` is one way to achieve this. In `ConcurrencyExample.java`, comment the bad-practice code and uncomment the good-practice code, then observe the execution results.

In addition, you could also use the APIs provided in `javafx.concurrent` package to enable the communication between background threads and the JavaFX Application thread. See here for a further introduction on this package.

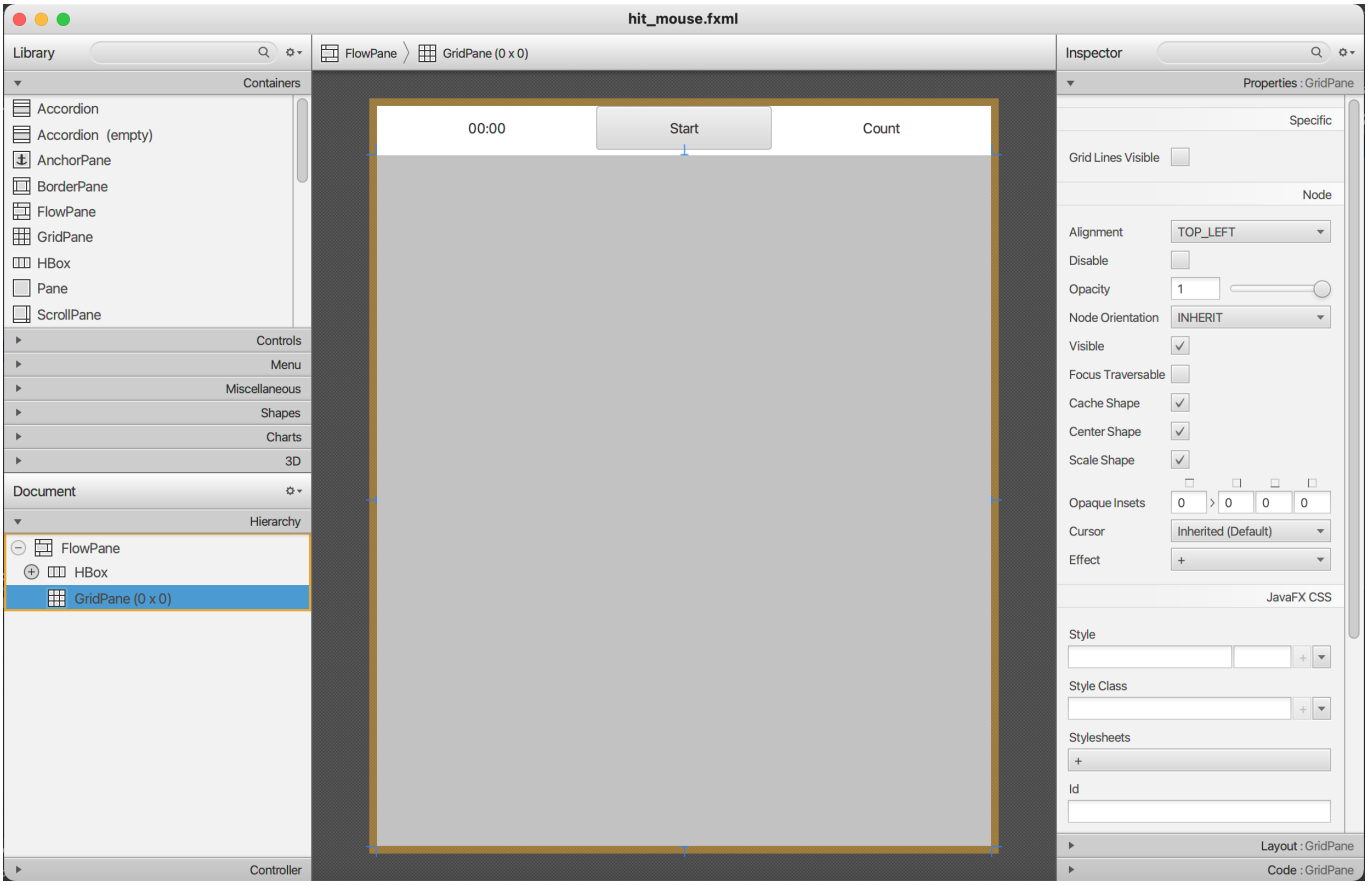
JavaFX - Whack-a-mole Game

The above example is slightly simple. In order to demonstrate multithreading more comprehensively, this section designed a `Whack-a-mole` game.

The initial UI of the game is shown below：


Whack-a-mole
00:00
Start
Count

The UI can be designed using Scene Builder:



Corresponding fxml:

```
<FlowPane fx:controller="com.example.whackamole.HitMouseController"
  xmlns:fx="http://javafx.com/fxml/1" alignment="center" hgap="5"
vgap="5">
  <HBox fx:id="hbHead" prefWidth="560" prefHeight="40">
    <Label fx:id="labelTime" prefWidth="200" prefHeight="40"
alignment="center" text="00:00" />
    <Button fx:id="btnStart" prefWidth="160" prefHeight="40"
alignment="center" text="Start" />
    <Label fx:id="labelCount" prefWidth="200" prefHeight="40"
alignment="center" text="Count" />
  </HBox>
  <GridPane fx:id="gpGrass" prefWidth="560" prefHeight="630" />
</FlowPane>
```

Note: the above layout file specifies the `HitMouseController`, which needs to be supplemented with business code. Before we start coding, there are four situations you need to consider about Whack-a-mole:

- the game starts with the holes empty.
- after a while, the mice drilled out of the hole
- the player hits a mouse
- the player hits the hole (the mouse has escaped)

For the above 4 situations, 4 types of holes and the corresponding 4 images need to be designed.

```
    private final static int TYPE_HOLE = 1; // hole
    private final static int TYPE_MOUSE = 2; // mouse
    private final static int TYPE_MOUSE_HIT = 3; // hit the mouse
    private final static int TYPE_HOLE_HIT = 4; // hit the hole
    private static Image imageHole; // hole image
    private static Image imageMouse; // mouse image
    private static Image imageMouseHit; // image of hit mouse
    private static Image imageHoleHit; // image of hit hole
    static {
        imageHole = new
Image(HitMouseController.class.getResourceAsStream("hole.png"));
        imageMouse = new
Image(HitMouseController.class.getResourceAsStream("mouse.png"));
        imageMouseHit = new
Image(HitMouseController.class.getResourceAsStream("mouse_hit.png"));
        imageHoleHit = new
Image(HitMouseController.class.getResourceAsStream("hole_hit.png"));
    }
```

Next, you need to initialize the controls on the UI, including the timer, clear the hit count, and the registration of the event for the start button, etc. You can add 4*5 buttons to the grid pane. Note that each button represents a hole, and each button registers a click event, indicating that the player will hit the hole.

It is the code that initializes each control:

```
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        // Initialization after the UI is opened
        // initialize each hole button and set the click event for each
hole button
        for (int i = 0; i < btnArray.length; i++) {
            for (int j = 0; j < btnArray[i].length; j++) {
                btnArray[i][j] = getHoleView(); // get a Hole button
                Button view = btnArray[i][j];
                gpGrass.add(view, j, i + 1); // add the hole button to the
grass grid
                int x = i, y = j;
                // Sets the action event for the hole button.
                // Clicking a hole in the ground means swinging the hammer
to whack a mouse
                // default kick the hole
```

```
                view.setOnAction(e -> doAction(x, y, TYPE_HOLE_HIT));
            }
        }
        labelTime.setFont(Font.font("KaiTi", 25));
        btnStart.setFont(Font.font("KaiTi", 25));
        labelCount.setFont(Font.font("KaiTi", 25));
        btnStart.setOnAction(e -> {
            // event handler after clicking the start button
            isRunning = !isRunning;
            if (isRunning) { // if the game state is running
                btnStart.setText("Stop");
                hitCount = 0; // clear hitCount
                timeCount = 0; // clear timeCount
                beginTime = new Date().getTime(); // get the beginning
time
                new MouseThread(0).start(); // start the first mouse
thread
                new MouseThread(timeUnit * 1).start(); // start the second
mouse thread
                new MouseThread(timeUnit * 2).start(); // start the third
mouse thread
            } else { // game over
                btnStart.setText("Start");
            }
        });
    }
```

The game sets that there will only be three mice to drill out of a hole at the same time, so when you click the start button, only 3 mouse threads are started. The mouse thread has to add the following processing logic:

(1) the mouse chooses which hole to drill out is completely random.

(2) Only choose empty holes

(3) The mouse should stay for a few seconds after coming out of the hole so that the player has enough time to beat it.

```
    private class MouseThread extends Thread {

        private int mDelay; // Delay interval

        public MouseThread(int delay) {
            mDelay = delay;
        }

        public void run() {
            try {
                sleep(mDelay); // different mouse has different delay
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            while (isRunning) { // the game state is running
```

```
                int i = 0, j = 0;
                while (true) {
                    // Randomly generate the position where the mouse
appears
                    i = new Random().nextInt(btnArray.length);
                    j = new Random().nextInt(btnArray[0].length);
                    if (timeArray[i][j] == 0) {
                        //do some action when the mouse go out the hole
                        doAction(i, j, TYPE_MOUSE);
                        break;
                    }
                }
                long nowTime = new Date().getTime();
                timeCount = (int) ((nowTime - beginTime) / 1000);
                try {
                    sleep((timeUnit - 100) * 3); // the time of the mouse
stay out the hole
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

Then write the `doAction` method to deal with 4 situations:

```
    // do some actions when the hole changes its state
    private synchronized void doAction(int i, int j, int type) {
        timeArray[i][j] = 3; // The mouse will stay out the hole for 3
seconds
        Button btn = btnArray[i][j];
        if (type == TYPE_HOLE_HIT) {
            showView(btn, imageHoleHit); // Show image of hit hole
            timeSchedule(i, j); // The hole timer began to count down
        } else if (type == TYPE_MOUSE) {
            showView(btn, imageMouse); // Show mouse image
            timeSchedule(i, j); // The hole timer began to count down
            btn.setOnAction(e -> { // Register the click event for the
hole button
                doAction(i, j, TYPE_MOUSE_HIT); // Once the mouse in the
hole is hit, do the TYPE_MOUSE_HIT action
                hitCount++; // Update hitCount
            });
        } else if (type == TYPE_MOUSE_HIT) {
            showView(btn, imageMouseHit);  // Show image of hit mouse
            btn.setOnAction(null); // Unregister the click event for the
hole button
        }
    }
```

In `showView` method, we use `Task`class in `javafx.concurrent` package. The Task class enables developers to implement asynchronous tasks in JavaFX applications. See here for a further introduction.

```
    private void showView(Button btn, Image image) {
        // define a JavaFX Task
        // The call method of a task cannot manipulate the interface;
        // the succeeded method does
        Task task = new Task<Void>() {

            // The thread inside the call method is not the main thread
            // and cannot manipulate the interface
            protected Void call() throws Exception {
                return null;
            }

            // The thread inside the succeeded method is the main thread
            //can manipulate the interface
            protected void succeeded() {
                super.succeeded();
                btn.setGraphic(new ImageView(image)); // Set the button
image as the input image
                labelCount.setText(String.format("Hit %d mice",
hitCount));
                labelTime.setText(String.format("%02d:%02d", timeCount /
60, timeCount % 60));
            }
        };
        task.run(); // start the JavaFX task
    }
```

Finally, after a hole was hit, whether hits the mounse or not, it will be restored to empty hole after a while.

TimerTask is used here, and the detailed implementation code is as follows:

```
    private void timeSchedule(int i, int j) {
        Button btn = btnArray[i][j];
        Timer timer = new Timer();
        timer.schedule(new TimerTask() { // The timer is scheduled once
per second
            public void run() {
                timeArray[i][j]--;
                if (timeArray[i][j] <= 0) { // time out
                    showView(btn, imageHole); // show empty hole
                    btn.setOnAction(e -> { // Registers the click event
for the hole
                        doAction(i, j, TYPE_HOLE_HIT);
                    });
                    timer.cancel(); // Cancel the timer
                }
            }
```

```
        }, 0, timeUnit);
    }
```

Reference:

- https://www.jetbrains.com/help/idea/javafx.html
- https://www.jetbrains.com/help/idea/opening-fxml-files-in-javafx-scene-builder.html#open-in-scene-builder
- https://jenkov.com/tutorials/javafx/concurrency.html
- https://github.com/aqi00/java/tree/master/chapter15/src/com/concurrent/mouse