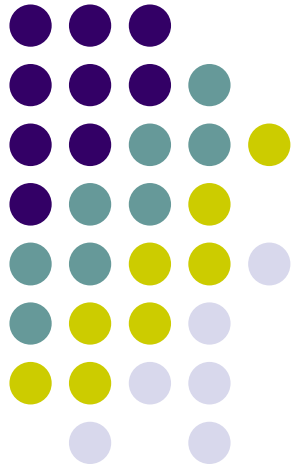
Setup JavaFX with JDK 15+

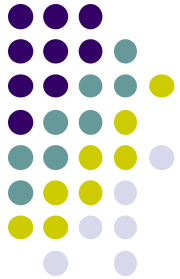
Downloads

JDK 15 Documentation

JavaFX Windows SDK SceneBuilder

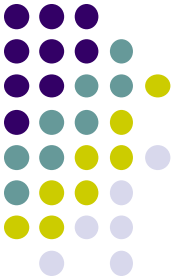


IntelliJ setup



Download the appropriate JavaFX SDK for your operating system and unzip it to a desired location, for instance

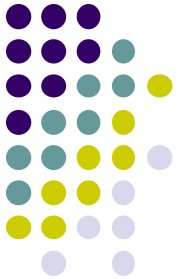
C:\Program Files\Java\javafx-sdk-15



IntelliJ setup

Define the JDK in IntelliJ IDEA

- Open the **Project Structure** dialog (e.g. Ctrl+Shift+Alt+S).
- In the leftmost pane, under Platform **Settings**, click SDKs.
- Above the pane to the right, click + and select **JDK 15**.
- In the dialog that opens, select the installation directory of the **JDK** to be used and click OK (C:\Program Files\Java\jdk-15)



IntelliJ setup

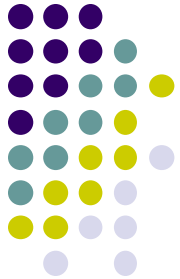
Setup SceneBuilder

- Open the Settings dialog (e.g. Ctrl+Alt+S).
- In the leftmost pane, under Platform **Languages&Frameworks**, click **JavaFX**.
- On the right side locate and set the path to the **SceneBuilder** executable.

By default it is found in

C:\Program Files\SceneBuilder

Modular JavaFX project with IJ



Required resources:

1. Download the Latest release of JDK 15 or later for the Windows operating system and unzip it to a desired location

`/Users/your-user/Downloads/jdk-15`

or

`C:\Program Files\Java`

2. Download the Latest release of JavaFX jmods for the Windows operating system and unzip it to a desired location, for instance

`/Users/your-user/Downloads/javafx-jmods-15`

or

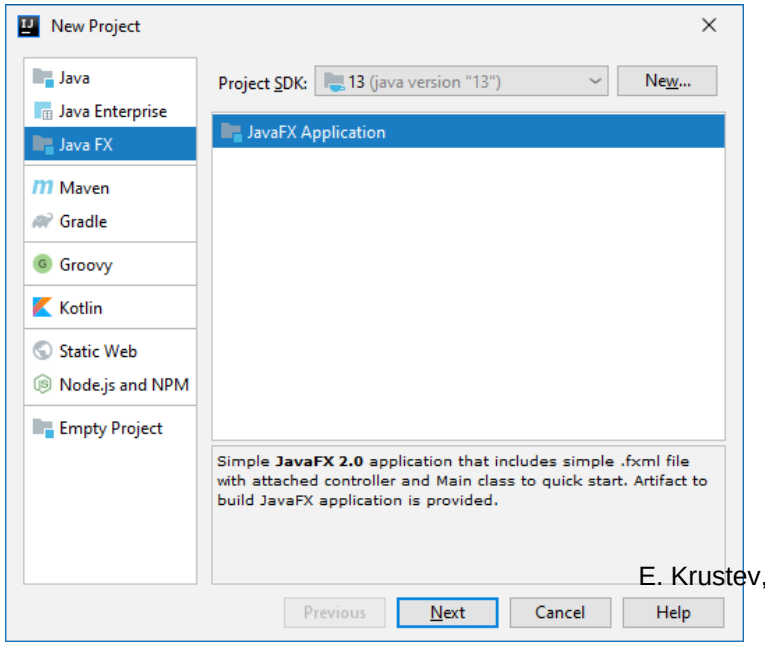
`C:\Program Files\Java`

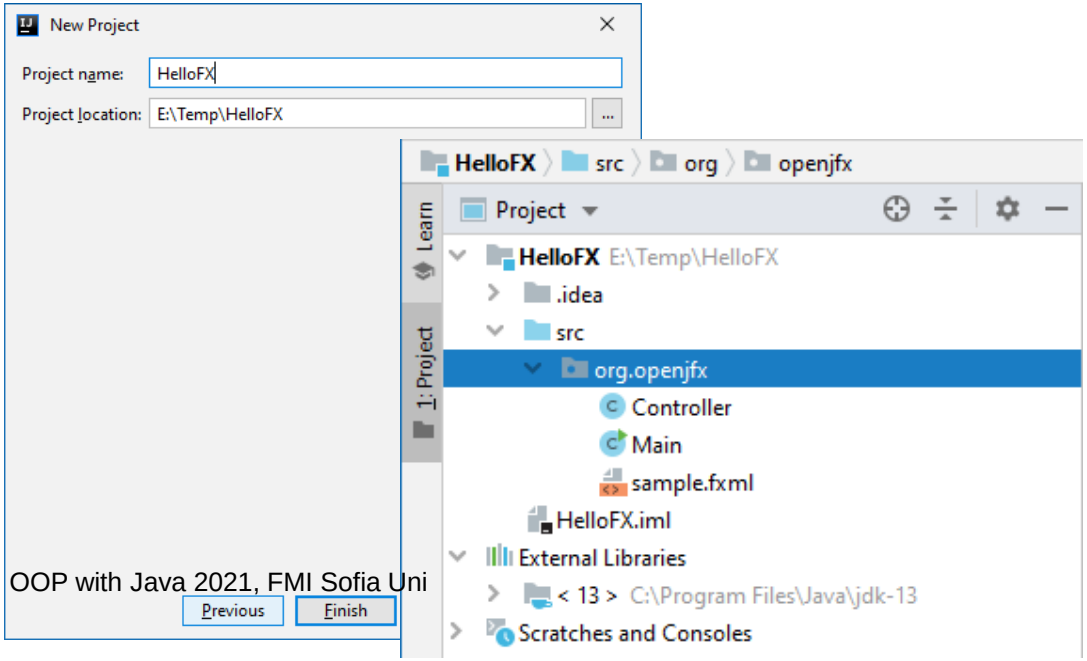
Modular JavaFX project with IJ

Follow these **steps** to **create a JavaFX modular project** and use the IntelliJ IDEA tools to build it and run it :

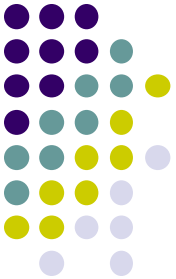
1. Create a JavaFX project

Provide a name to the project, like **`HelloFX`**, and a location. When the project opens, **rename** the **`hellofx`** package to **`org.openjfx`**



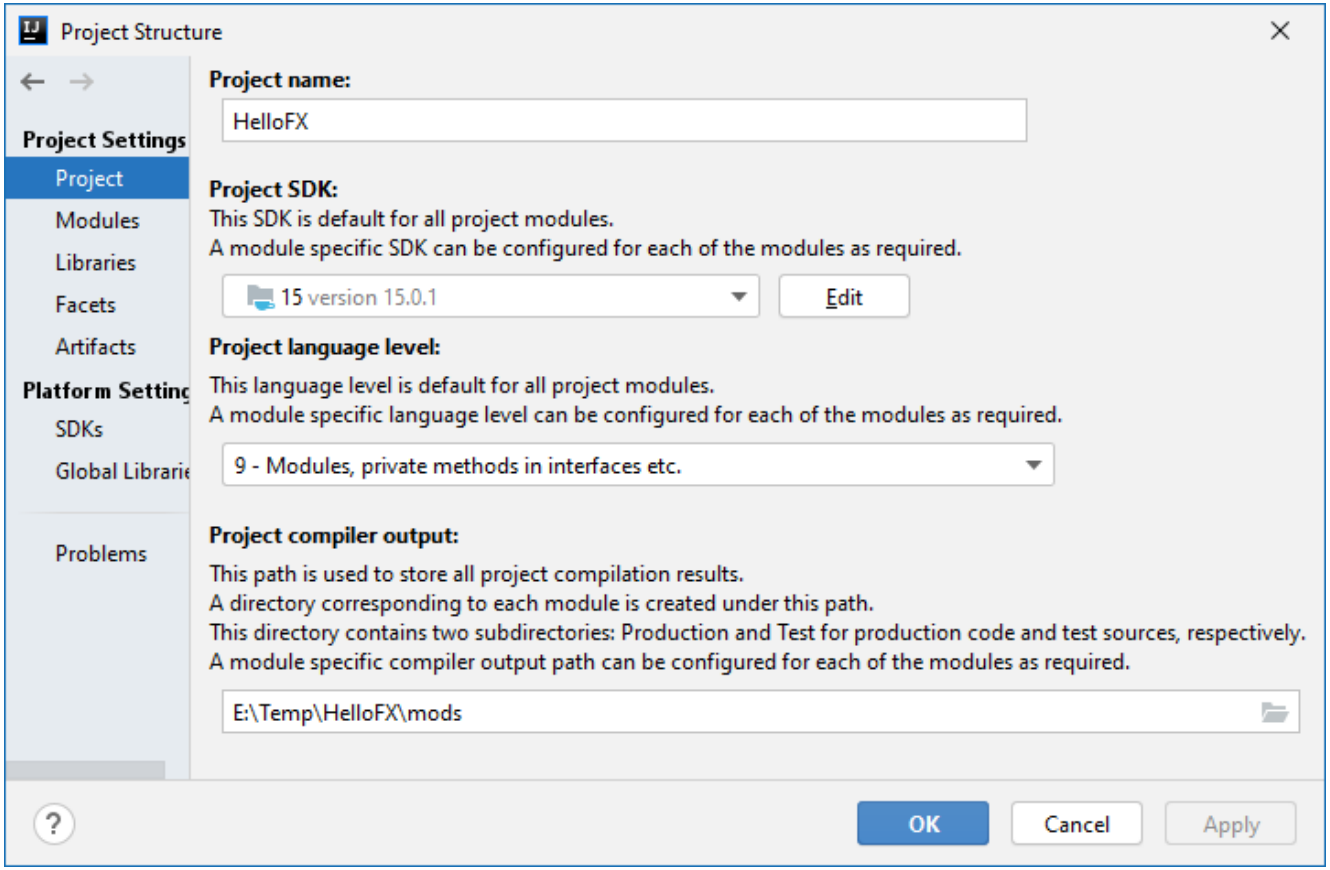


Modular JavaFX project with IJ



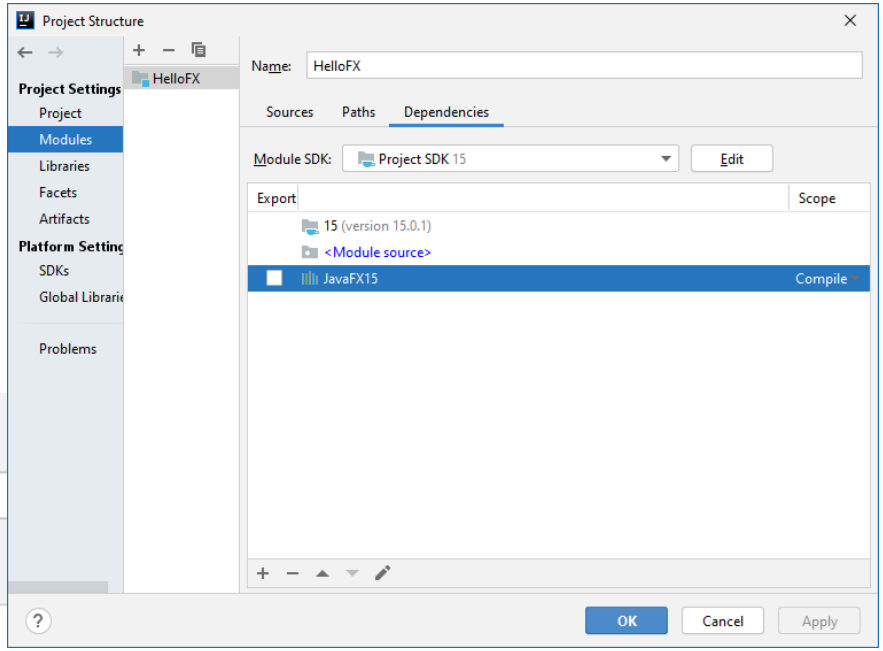
2. **Set JDK 13** and add **JavaFX 13**

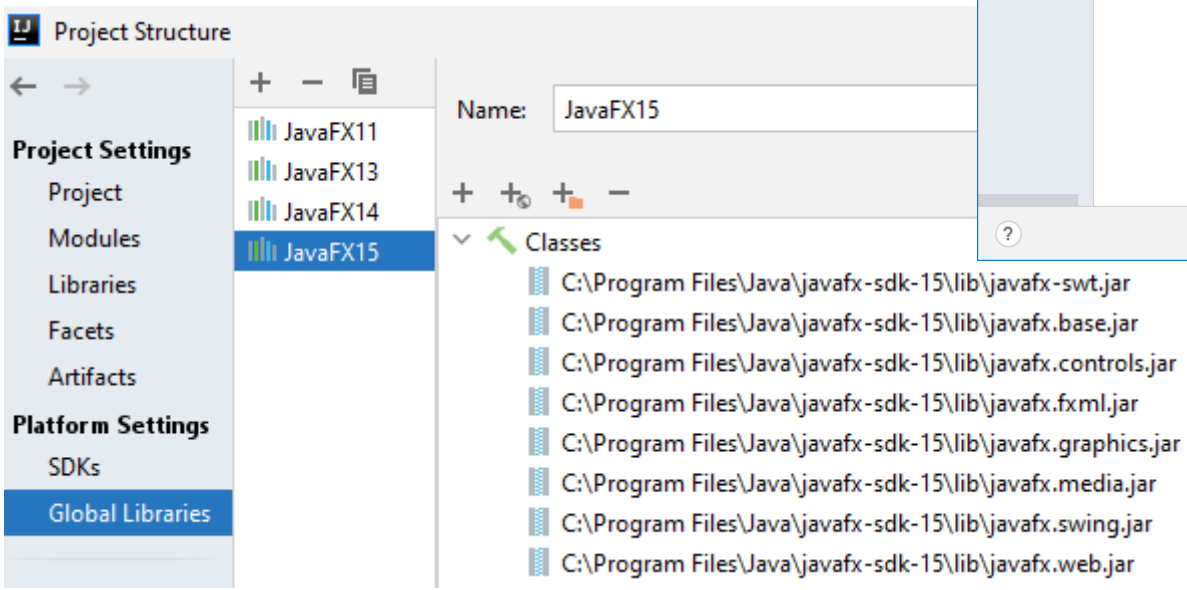
Go to
File ->
Project Structure ->
Project
and set the **Project SDK** to **JDK 13**.
Set the
Project language level to **`Modules, private methods in interfaces etc.`**
and change the default **compiler output directory** out to **`mods`**.



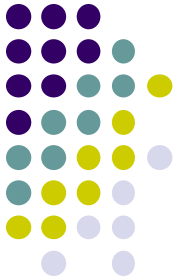
Modular JavaFX project with IJ

3. Go to File -> Project Structure ->Global Libraries to **create** (if not defined earlier by selecting the **`lib`** folder of the JavaFX SDK) and **add** the JavaFX 15 SDK **as a library to the project**.





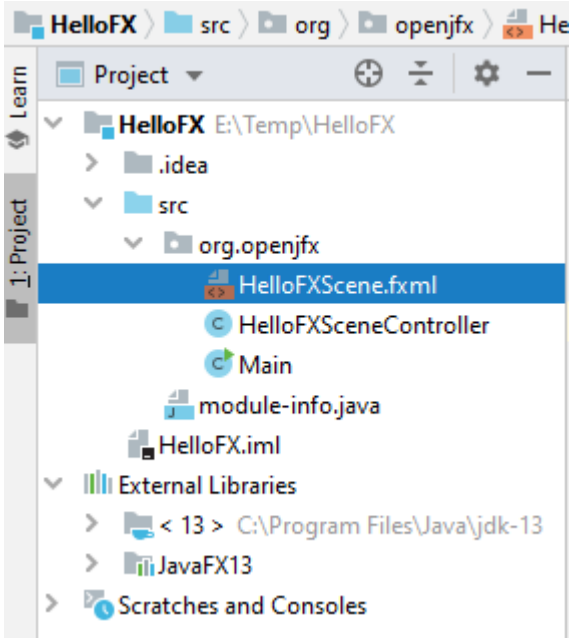
Modular JavaFX project with IJ



Note:

The Controller is the code- behind file for the FXML file. Therefore, name of the Controller is built from the name of the FXML file adding suffix Controller.

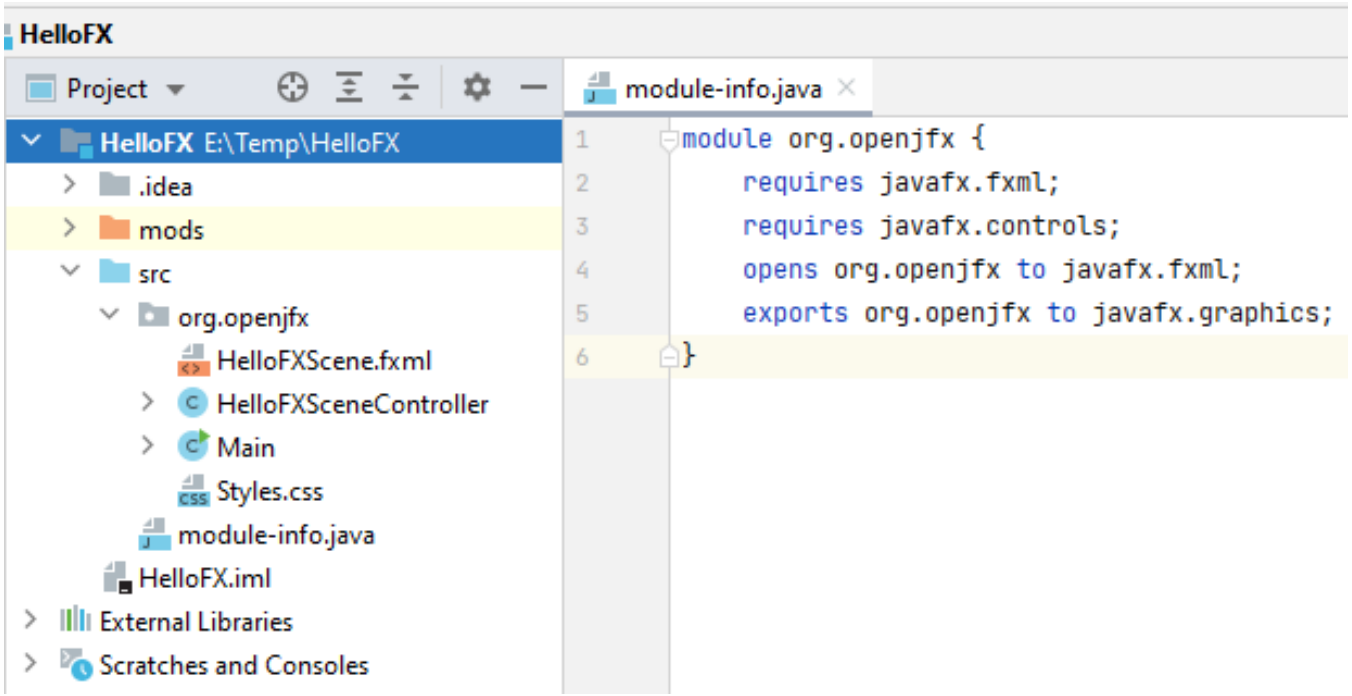
In this example rename the FXML file to **HelloFXScene** and accordingly **HelloFXSceneController** for its code- behind file.



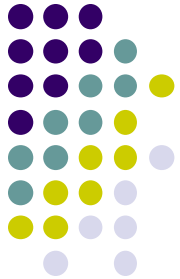
Modular JavaFX project with IJ

4. Add the **`module-info`** class.

Add the **`module-info`** class to package **`org.openjfx`**, including the required modules **`javafx.controls`** and **`javafx.fxml`**. Since FXML uses reflection to access the controller in the module, this has to be opened to **`javafx.fxml`**. Finally, **export** the package **`org.openjfx`**.



Modular JavaFX project with IJ



Hints:

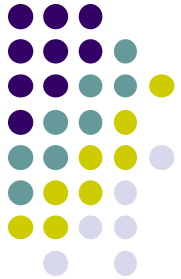
1. Use the name of the root named package for the name of the module
2. Highlight the given content for the **`module-info`** class

```
requires javafx.fxml;
requires javafx.controls;
opens $PACKAGE$ to javafx.fxml;
exports $PACKAGE$ to javafx.graphics;
```

and save it as a Live template (**Tools-**> **Save as Live template**)

Give the template abbreviation for example, **fx-fxml-info**, and reuse it whenever you need to create a FXML modular project.

Modular JavaFX project with IJ



5. Add the **`source code`**.

Making use of the sample **Main** class, add its content to the project **Application** class. Then **add** the **Controller** and the **FXML** and the **CSS** files.

```
public class Main extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("HelloFXScene.fxml"));

        primaryStage.setTitle("Hello JavaFX");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }

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

}
```

Modular JavaFX project with IJ

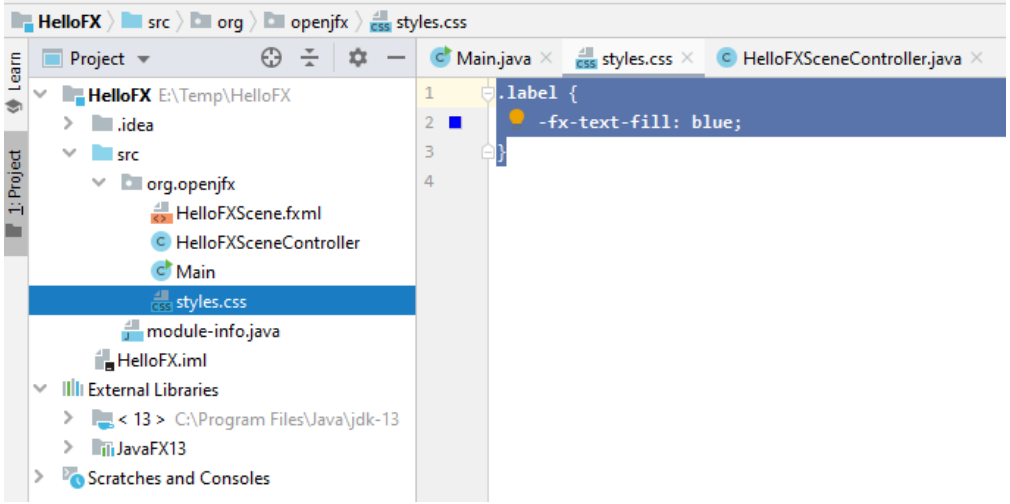
HelloFXSceneController.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.StackPane?>


<StackPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
           minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
           xmlns="http://javafx.com/javafx/8.0.171"
           xmlns:fx="http://javafx.com/fxml/1"
           fx:controller="org.openjfx.HelloFXSceneController">
    <children>
        <Label fx:id="label" text="Label" />
    </children>
</StackPane>
```

Modular JavaFX project with IJ

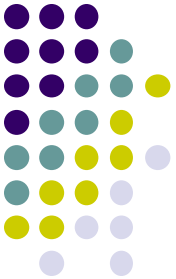
styles.css

```
.label {
    -fx-text-fill: blue;
}
```



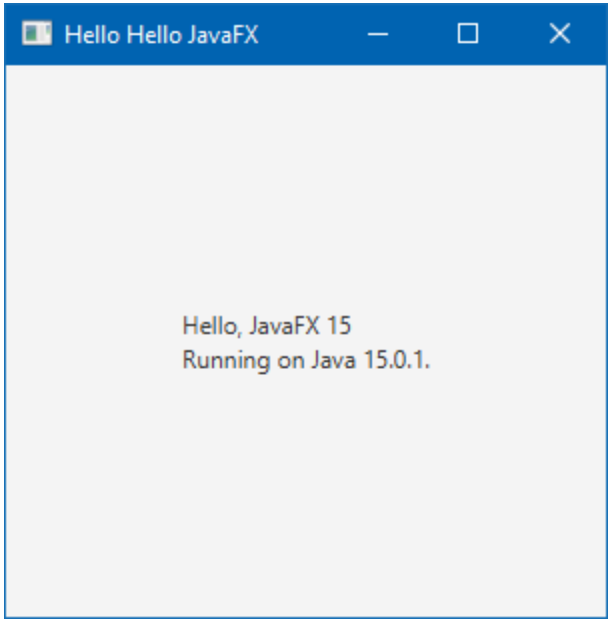

```
public class HelloFXSceneController {

    @FXML
    private Label label;

    public void initialize() {
        String javaVersion = System.getProperty("java.version");
        String javafxVersion = System.getProperty("javafx.version");
        label.setText("Hello, JavaFX " + javafxVersion + "\nRunning on Java “
                      + javaVersion + ".");
    }
}
```

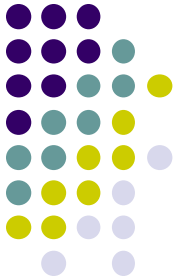
Modular JavaFX project with IJ



6. No need to add **`VM options`**.
7. Run the project

Click **Run -> Run...** to run the project.





Reflection is commonly used with dependency injection. One example of this is an FXML-based JavaFX application. (See "Define Custom Behavior in FXML with FXMLLoader"). When an FXML application loads, the controller object and the GUI components on which it depends are dynamically created as follows:

First, because the application depends on a controller object that handles the GUI interactions, the FXMLLoader injects a controller object into the running application. In other words, the FXMLLoader uses reflection to locate and load the controller class into memory and to create an object of that class.

Next, because the controller depends on the GUI components declared in FXML, the FXMLLoader creates the GUI controls declared in the FXML and injects them into the controller object by assigning each to the controller object's corresponding @FXML instance variable. In addition, the controller's event handlers that are declared with @FXML are linked to the corresponding controls as declared in the FXML.

Once this process is complete, the controller can interact with the GUI and respond to its events. We use the **`opens ... to`** directive to allow the FXMLLoader to use reflection on a JavaFX application in a custom module.