Configuring a JavaFX Project in IntelliJ IDEA

# Project Structure

When you create a new Java project in IntelliJ IDEA, you will see the following items in the left-hand pane, which is known as the **Project tool window**:

* ProjectName
* External Libraries
* Scratches and Consoles

### ProjectName

This is the directory of your Java project.

By default, upon creation this directory will contain two items:

* .gitignore (hidden plain text file)
* src (subdirectory)

**`.gitignore`**

`.gitignore` is a plain text configuration file. It specifies which files and directories should be ignored by Git. Changes to files listed in this configuration file will not be tracked, nor included in Git commits.

*\*Since .gitignore it starts with the `.` character it won't be visible in Finder unless you have configured Finder to display hidden files. It will show up in the Project Tool window in IntelliJ.*

**`src`**

The `src` directory is where the source code for the project is stored. The `src` directory is used for organisation purposes, so that the source code is separated from other project files and directories, such as such as configuration files, build outputs, documentation and resources.

Typically this directory contains the `Main.java` file containing the `main` method (entry point of a Java application), as well as user-defined **package subdirectories**. However, there is flexibility in how the source code can be organised within the `src` directory.

**Java Packages (Namespaces)**

In Java, a namespace is called a `package`. There is no `namespace` keyword in Java like in C# or C++, instead, the keyword `package` is used.

To put any classes in a package, the following line should be added to the beginning of the `.java` file:

package projectname.packagename;

This puts everything in the file into the `projectname.packagename` namespace.

There are a couple of important points regarding the use of packages:

* There can only be one package declaration in a single file.
* The package declaration must come before any other code in the file.

Not doing either of the above will cause a compilation error.

**Organisation of Java Packages**

In a Java project, user-defined packages are typically stored in subdirectories of the main project folder. In IntelliJ IDEA, you can add a new package by right clicking on the `src` folder in the Project Tool window and clicking `New` -> `Package`.

To explicitly mark any Java code as belonging to this namespace, you MUST put the package declaration in the relevant files, otherwise any classes within these files will be treated as part of the default namespace.

**Importing Java Packages**

The following syntax is used to import a package in Java:

import packagelocation.packagename;

When you import a package in a Java file, the Java compiler and runtime environment (JRE) look for the specified package in the **classpath**. The classpath is a list of directories, JAR files and ZIP files that the compiler and JVM search for classes and resources. By, default the classpath includes the current directory.

If you are using an IDE, the IDE will manage the classpath for you.

**Important Note Regarding the Creation and Importation of Packages**

It is crucial when creating and importing packages that you do BOTH of the following steps:

1. Create a subdirectory within the classpath with the package name.
2. Add a `package` declaration to any `.java` files within this directory.

Not creating a subdirectory with the package name within the classpath will result in the compiler not being able to locate the package and its classes, causing a `package does not exist` error during compilation.

Not including the `package` declaration in the `.java` files would cause the classes within the files to be treated as part of the default package. This would lead to namespace confusion and potential conflicts.

### External Libraries

This includes all the external libraries and dependencies that your project uses. This includes the JDK, any JavaFX libraries, and other third-party libraries that are part of your project's classpath.

### Scratches and Consoles

This is a special area in IntelliJ IDEA where you can create temporary files, notes, and scripts that are not part of your main project.

While the `ProjectName` listing in the Project tool window refers to the specific directory of your Java project, `External Libraries` is a logical grouping created by the IDE to show the dependencies of the project, and `Scratches and Consoles` is an internal feature of IntelliJ IDEA.

# Adding JavaFX Libraries to Project

Starting from JDK 11, the JavaFX SDK is not included in the default JDK. Therefore these libraries need to be added separately to JavaFX projects.

It is common practice to put all Java-related libraries (such as the JDK and JavaFX SDK) in the following directory (on macOS):

/Library/Java/JavaVirtualMachines

The following are the steps to add the JavaFX SDK to an IntelliJ IDEA project:

1. Go to `File`-> `Project Structure`
2. On the left side of the `Project Structure` window, there is a `Project Settings` menu. Click the `Libraries` tab from this menu.
3. Click the `+` sign to add a new library. Then click `Java`, as we are adding a Java library.
4. Navigate to the JavaFX SDK directory, select the `lib` folder within this directory and click `Open`. Then, click `OK`.

## JavaFX SDK

The JavaFX SDK contains the following items:

* **JavaFX API**: an extensive set of libraries of packages for building rich graphical user interfaces.
* **JavaFX Scene Builder**: a standalone, IDE-independent, executable application that provides a visual layout tool for designing JavaFX application user interfaces. It generates FXML code based on the layout created.
* **JavaFX Runtime Environment**: the JavaFX runtime environment is an essential component of the JavaFX SDK that provides the necessary environment for executing JavaFX applications. It includes various libraries and components that facilitate the running of applications developed using the JavaFX API.

The JavaFX Runtime Environment consists of the following:

* **JavaFX Libraries**: a set of `.class` (bytecode) files containing the libraries of the JavaFX API. The purpose of including these in the runtime environment (which will exist not only on a developer's computer but also on the machines of users of the application) is due to how dependencies are handled in Java. To avoid the need to recompile JavaFX packages in every JavaFX application, these are stored once on a user's machine. By avoiding the need to recompile these dependencies every time they are included in a JavaFX application, the file size of Java Archive (JAR) files is reduced. Furthermore, updates to the code encapsulated in the JavaFX packages can be used in all JavaFX applications (as this is called from outside the main program source code), and they do not require recompilation to achieve this.
* **JavaFX Media Engine**: part of the runtime is the media engine, which enables playback of audio and video content within JavaFX applications. This engine supports various media formats and provides capabilities for controlling media playback.
* **Web Engine and WebView**: the runtime includes a web engine that allows developers to embed web content within JavaFX applications using the WebView component. This supports modern web standards like HTML5, CSS and JavaScript.
* **Hardware Acceleration**: the JavaFX runtime leverages hardware acceleration to improve the performance of graphical rendering.
* **Support for FXML**: the runtime supports FXML, an XML-based language for defining the user interface layout.
* **Sample Applications and Documentation**: a set of example projects and comprehensive documentation to help developers learn and utilise JavaFX features effectively.

# Configuring VM Options for JavaFX

Configuring VM options for JavaFX involves setting specific parameters in the Java Virtual Machine to include the JavaFX library path and required modules, ensuring that the JavaFX application can run properly.

If you try to run a JavaFX application without having configured the JVM to do so, you will likely encounter the following error:

*Error: JavaFX runtime components are missing, and are required to run this application*

To configure the JVM for running JavaFX applications, follow the steps below:

1. Open the project in IntelliJ IDEA
2. Click `Run` - this is the menu in IntelliJ that contains options related to running your project and configurations.
3. Click `Edit Configurations` - this opens the configuration settings for how your project will be executed. Here, you can specify details like the main class to run, VM options, environment variables etc.
4. Click the `+` button to add a new **run configuration**. A run configuration is a set of settings that define how to execute a program. It includes details like:

* The main class to run
* Program arguments and environment variables
* VM options
* JRE to use
* Working directory

A run configuration needs to be created specifically for the JavaFX project.

1. In the list of options that appears after clicking the `+` sign, select `Application`. This specifies that you are creating a run configuration for a standard Java application, which includes JavaFX applications.
2. Select the `Main` class (or whichever class you are using as the program entry point containing the `main` method) and name the configuration the same as this class.

For simple command-line applications, IntelliJ can often auto-detect the main method by scanning the project for classes with the `public static void main(String[] args)` signature. However, in larger projects the entry point needs to be explicitly defined.

1. Click `Modify Options` - this button allows you to add additional options to the configuration.
2. Click `Add VM Options` - virtual machine options are passed to the JVM when the application starts. The options we are adding are necessary for running JavaFX applications.
3. In the text field for VM options, add the following line:

*--module-path /Library/Java/JavaVirtualMachines/javafx-sdk-22.0.1/lib --add-modules javafx.controls,javafx.fxml*

A **module** in Java is a collection of packages and resources. The **module path** is a path that specifies where the Java runtime can find the required modules.

The core JavaFX modules are **`javafx.controls`** and **`javafx.fxml`**. These modules cover the most common JavaFX functionalities. They include the core packages such as `javafx.application` and `javafx.scene` etc.

The first part of the line specifies the module path to the JavaFX SDK.

The second part specifies to add the core JavaFX modules (`javafx.controls` and `javafx.fxml`) to the application.

# Sample Project with Detailed Explanation

1. import javafx.application.Application;

2. import javafx.scene.Scene;

3. import javafx.scene.control.Label;

4. import javafx.scene.layout.StackPane;

5. import javafx.stage.Stage;

6.

7. public class Main extends Application {

8.

9. @Override

10. public void start(Stage primaryStage) {

11. Label label = new Label("Hello, JavaFX!");

12. StackPane root = new StackPane(label);

13. Scene scene = new Scene(root, 400, 300);

14.

15. primaryStage.setTitle("JavaFX Example");

16. primaryStage.setScene(scene);

17. primaryStage.show();

18. }

19.

20. public static void main(String[] args) {

21. launch(args);

22. }

23. }

24.

**`javafx.application.Application`**

The `javafx.application` package contains classes and interfaces that provide the basic framework for a JavaFX application. The most important class in this package is the `Application` class, which provides the lifecycle methods (`init()`, `start()`, `stop()`, these are for initialisation and cleanup when starting or exiting the application), and the `launch` method to start the JavaFX application.

**`Application` Class**

The `Application` class is an abstract class in the `javafx.application` package. It provides the foundational structure for a JavaFX application.

It is common practice for the main application class (in this example `Main`) to extend the `Application` class. This is so that lifecycle methods of the `Application` class such as `init()` and `start()` can be overridden in order to customise them to the specific application.

Since we are talking about the main application class, it will also contain the `main` method (program entry point). The body of this method will contain the `launch(args)` method call, which starts the JavaFX application by invoking the `start` method.

**Stage**  
The term "stage" refers to a window in the GUI. A stage is represented by the `Stage` class from the `javafx.stage` package.

A `Stage` represents a window in which your application is displayed (but not the contents of that window). A **primary `Stage`** is provided by the `start` method in `Application`. Closing the primary stage exits the application.

Key points about `Stage`:

* Represents a window of the application
* You can set properties like title, size and modality
* It contains a `Scene`, which represents the content displayed in the window.
* You can have multiple stages in a JavaFX application, such as popup windows or dialogs.

***\*A `Stage` represents a window itself, while a `Scene` represents the contents of that window.***

**Scene**

A `Scene` contains all the UI elements that make up the user interface within a `Stage`. These elements are arranged in a tree structure known as a **scene graph**. The tree structure of UI elements in a JavaFX scene graph differs from a binary tree in that while a binary tree can have up to 2 children per parent node, there is no limit to the number of children a node in a scene graph can have.

UI elements within a scene are referred to as nodes.

**Layout Panes**

A layout pane in JavaFX is a container node that is very similar in function to "stacks" (such as ZStack, VStack and HStack) in SwiftUI. Layout panes provide different layout policies and rules to organise their children in various ways.

Types of layout panes include:

* `StackPane` - stacks all children on top of each other.
* `VBox` - arranges children vertically in a single column.
* `HBox` - arranges children horizontally in a single row.
* `GridPane` - arranges children in a grid of rows and columns.
* `BorderPane` - lays out children in five regions: top, bottom, left, right, and centre.

**`import` Statements**

1. The `Application` class, which is contained in the `javafx.application` package provides the `launch` method to start the application and the `start` method to set up the primary stage (window).
2. The `Scene` class, contained in the `javafx.scene` package, represents the container for all content in a scene graph.
3. The `Label` class from the `javafx.scene.control` package is a UI control that displays a short text string or an image.
4. The `StackPane` class from the `javafx.scene.layout` package is a layout container that stacks all its children on top of each other.
5. The `Stage` class from the `javafx.stage` package represents a top-level window with a title and a border.

## Program Initialisation

The `main` method (entry point of the program) calls the **`launch`** method.

**`launch(args)`**

The `launch` method is a static method of the `Application` class. It does the following:

1. Initialises the JavaFX runtime environment.
2. Creates an instance of the class that extends `Application` (in this case, the `Main` class). This is done internally by the JavaFX framework, and does not require explicitly creating an instance of the class using `new Main();`.
3. Calls this class's `init` method which may be overridden for initialisation (by default does nothing).
4. Calls this class's `start` method to begin the application's lifecycle.

*When referring to "beginning the application's lifecycle" in the context of a JavaFX application, this specifically means the lifecycle (execution flow) managed by the JavaFX framework, which is distinct from the basic execution of the Java program that begins with the `main` method.*

When you run a Java program, the JVM starts and begins executing the `main` method. This is the entry point of the program. Inside the `main` method, when you call `launch(args)`, you are transitioning from the general Java execution flow to the JavaFX-specific execution flow, but it is still within the same JVM process.

**`start` Method**

The `start` method in the `Application` class is an abstract method. This requires the `Main` subclass to provide an implementation for the method.

The purpose of the `start` method is to build and configure the JavaFX UI.

The signature of the `start` method is:

public abstract void start(Stage primaryStage)

The `primaryStage` parameter represents the primary window of your application. It is passed to the `start` method from within the `launch` method where it is called.

Within the `start` method, you would typically do the following:

* Create the root node (e.g., `StackPane`, `BorderPane`)
* Create and configure the scene with this root node (pass the root node as an argument to the `Scene` constructor, making it the base of the scene graph).
* Set the scene on the primary stage.
* Show the primary stage using `primaryStage.show()`.

This is done as follows (pasted from the example above):

11. Label label = new Label("Hello, JavaFX!");

12. StackPane root = new StackPane(label);

13. Scene scene = new Scene(root, 400, 300);

1. A new `Label` object is created. The string passed to its constructor will be displayed by the label.
2. A new `StackPane` object is created (like a SwiftUI ZStack). Passing the label to its constructor adds the label to the StackPane and will be displayed within it.
3. A new `Scene` object is created. This `Scene` represents what will be the content area of the primary stage (window).

*The `Scene` is essentially a container for the scene graph (hierarchical tree structure of all UI elements in the scene).*

The constructor of the `Scene` class takes the root node of the scene graph and the width and height of the scene as arguments.

The size of the stage is generally dependent on the size of its scene. When you set a scene on a stage, the stage will adjust its size to accommodate the dimensions of the scene.

15. primaryStage.setTitle("JavaFX Example");

16. primaryStage.setScene(scene);

17. primaryStage.show();

1. This sets the title of the primary stage to "JavaFX Example". The title appears in the title bar of the window.
2. This assigns the `Scene` object (created in line 13) to the primary stage.
3. This renders the scene and all its nodes so that the user can see and interact with the UI components.