CS3231
Object Oriented Programming I

Name: ________________________________ (   ) Date: _______________

Chapter 13: Introduction to JavaFX

Lesson Objectives

After completing the lesson, the student will be able to

- *appreciate the usefulness of GUI programming*
- *write a simple JavaFX program and understand the relationship among stages, scenes, and nodes*
- *create simple user interfaces using panes and simple UI controls*

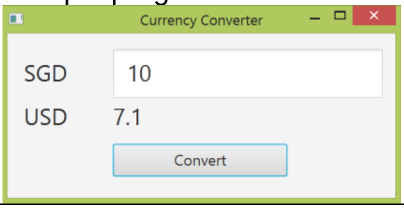
13.1 What is GUI Programming?

Most modern applications, whether designed for desktop, web, or enterprise, use a graphical user interface (GUI, pronounced goo-ey) to interact with the user. So far, all of the programs you have dealt with and coded are console based programs. Console applications use text-based program for output and keyboard entry for user input. A GUI based application make use of widgets to interact with users. A widget is an element of a graphical user interface (GUI) that displays information or provides a specific way for a user to interact with the operating system or an application.

Widgets include icons, pull-down menus, buttons, selection boxes, progress indicators, on-off checkmarks, scroll bars, windows, window edges (that let you resize the window), toggle buttons, form, and many other devices for displaying information and for inviting, accepting, and responding to user actions.

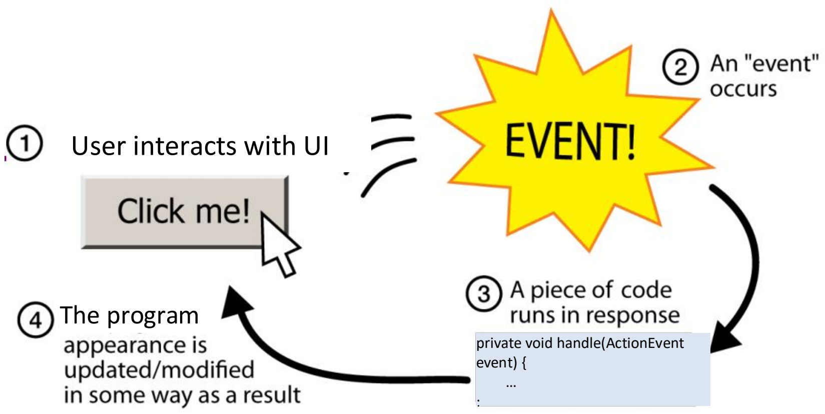
Interactive programs can be classified into two types:

Program-Driven = Proactive	**Event-Driven = Reactive**

• Statements execute in sequential, predetermined order
• Typically use keyboard or file I/O, but program determines when that happens
• Usually single-threaded
 	
Sample program:
`Now enter the input in SGD Dollars >> $10`
`SGD$10.0 = USD $7.84`
`Thank you for using the currency converter`
	• Program waits for user input to activate certain statements
	• Typically uses a GUI (Graphical User Interface)
	• Often multi-threaded
	
	Sample program:
	
	Currency Converter
	SGD 10
	USD 7.1
	Convert

With the development of a program with a graphical user interface (GUI) normally you want something to happen when the user performs certain actions, for example, clicking a button. This type of program is referred to as an event-driven program.

Event-driven programming is a programming style that uses a signal-and-response approach to programming. In event-driven programming, code is executed upon activation of events.


1 User interacts with UI
Click me!
2 An "event" occurs
EVENT!
3 A piece of code runs in response
4 The program appearance is updated/modified in some way as a result

```
private void handle(ActionEvent
event) {
      ...
:
```

13.2 Why JavaFX?

In the beginning, there was AWT or Java's abstract window toolkit which was used for both Applets (embedded Java programs on webpages) and Applications. With this you could create buttons, checkboxes, text fields, etc and make a program with a basic user interface.

Then came Swing, Java's updated GUI (graphical user interface) library with more tools and an updated look and feel. Swing is still very popular and widely used, but now that Java has Oracle at the helm we are once again seeing Java move into a new and exciting direction JavaFX!

If you go directly to the JavaFX faq source at Oracle they state clearly that "JavaFX is replacing Swing as the new client UI library for Java" which is why it makes sense as Java developers that we start taking JavaFX seriously and start embracing it as the best way to build applications in Java. Here's a quick summary of some useful/powerful features of JavaFX:

Provides a powerful Java-based UI platform capable of handling large-scale data-driven business applications.

Completely developed in Java and leverage on the power of standards-based programming practices and design patterns.

Provides a rich set of UI controls, graphics and media API with high-performance hardware-accelerated graphics and media engines to simplify development of immersive visual applications.

Consistent and modern API design

Integration with JavaFX binding and collection
- Concept of binding and observable properties/ collection.

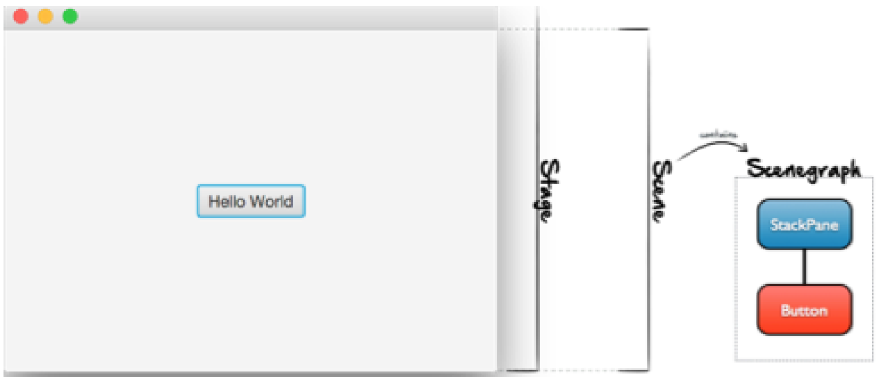
Powerful CSS styling
- CSS allows you to reach every node in the user interface and to completely change the style of what is shown to the user.
- Two separate looks for JavaFX (Caspian & Modena)

For more features, refer to:

http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html

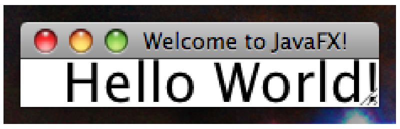
13.3 Anatomy of a JavaFX application

The following diagram shows the structure of a JavaFX application.

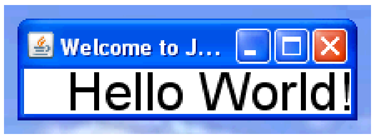


Stage is the top-level container. A Stage can either be represented as a Frame for desktop applications or a rectangular region for applications embedded in a browser. You can define and skin the main window of the JavaFX application through the Stage's properties and methods. For example:

Mac OSX:



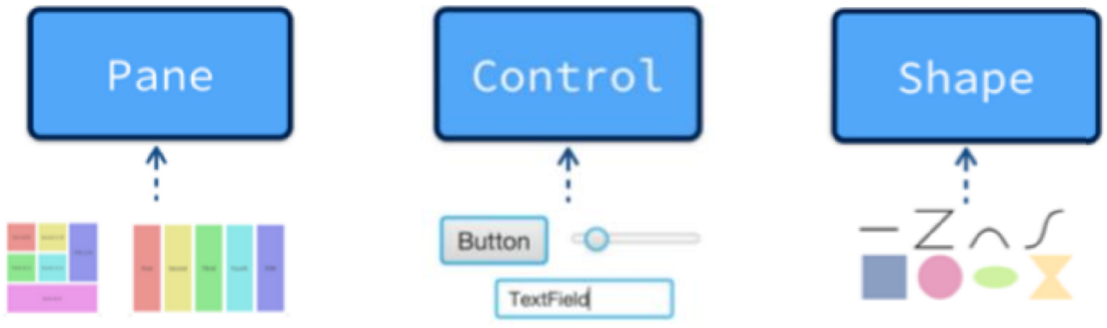
Windows XP:



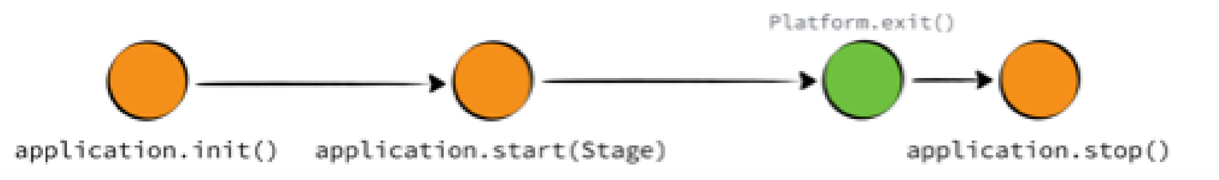
Scene is the container for the visual content of the Stage. Think of watching a play (on a stage) and when something is happening you are watching a scene. When the story changes, the play goes to a NEW scene. In the program when you want to switch to another view you can also have the Stage change to a different Scene.

The Scene's content is organized in a **Scene graph**. Scene graph is a hierarchical tree of nodes that represents all of the visual elements of the application's user interface. A single element in a scene graph is called a **node**.

Some example of nodes includes:



A JavaFX application runs in the default JavaFX lifecycle that is defined by the Application class.



Now that we have a better understanding of the structure of a JavaFX program, let's start building a simple JavaFX HelloWorld program. There are two different approaches to coding a JavaFX program:

1. **Using pure code:** In this technique you are writing ALL the code to create the interface (yes, every label and button you wish to add in the program!) AND the logic of the interface, both in the same class.

2. **Using FXML and SceneBuilder:** FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code. Using FXML and then using Scene Builder as a WYSiWYG editor is a best practice when developing views for a JavaFX application.

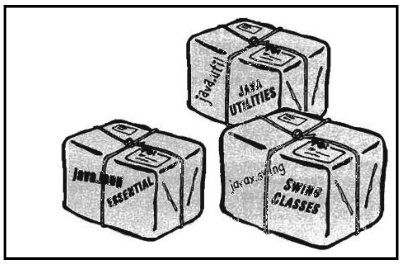
We will cover both approaches in this module.

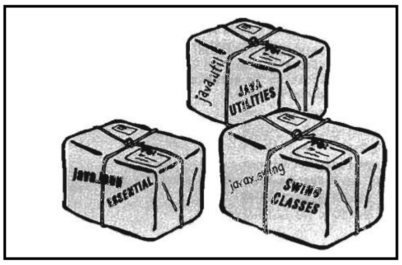
13.4 Understanding JavaFX 13 and Modules

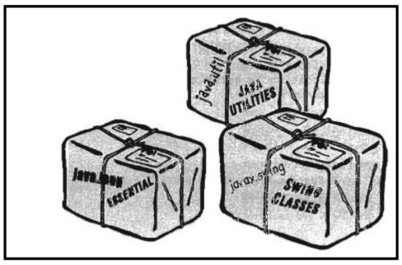
Before we start coding our first JavaFX program, let's first take a look at JavaFX 13 and Java's Module Structure.

In Chapter 8, we learnt that Java contains many predefined classes that are grouped into categories of related classes called **packages**. From JDK 9 onwards, packages are further grouped in **modules**. Every class in the Java library belongs to a package. Together, these are known as the **Java Application Programming Interface (Java API)**, or the Java class library.

Related packages are then further grouped into modules.







Why do we need Java modularity?

Modularization of the JDK enables the source code to be completely restructured in order to make it easier to maintain.

Module is a named, self-describing collection of code and data. Its code is organized as a set of packages containing types, i.e., Java classes and interfaces; its data includes resources and other kinds of static information.

To control how its code refers to types in other modules, a module declares which other modules it requires in order to be compiled and run. To control how code in other modules refers to types in its packages, a module declares which of those packages it exports. **This is done via the module-info.java file, also known as the module descriptor.**

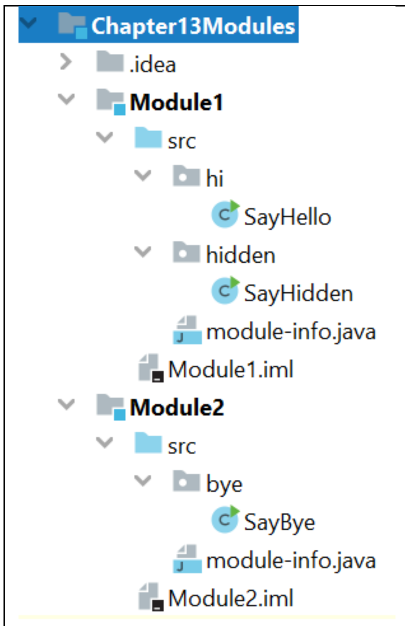
Modules offer stronger encapsulation than jars.

Modularity is a general concept. In software, it applies to **writing and implementing a program or computing system as a number of unique modules, rather than as a single, monolithic design.** A standardized interface is then used to enable the modules to communicate. Partitioning an environment of software constructs into distinct modules helps us minimize coupling, optimize application development, and reduce system complexity.

Modularity enables programmers to do functionality testing in isolation and engage in parallel development efforts during a given sprint or project. This increases efficiency throughout the entire software development lifecycle.

Let's take a look at an example.

Let's say that we have two modules in our application (i.e. the Project named Chapter13Modules):



```java
package hi;
public class SayHello {
    public static String s = "can see :)";
    public static void main(String[] args){
        System.out.println("hello");
        System.out.println(s);
    }
}
```

```java
package hidden;
public class SayHidden {
    public static String s = "can't see even if not export :(";
    public static void main(String[] args){
        System.out.println("hidden");
        System.out.println(s);
    }
}
```

```java
package bye;
import hi.SayHello;
//import hidden.SayHidden;
public class SayBye {
    public static String s = "can only see in bye >_<";
    public static void main(String[] args){
        System.out.println("bye");
        System.out.println(s);
        System.out.println(SayHello.s);
        //System.out.println(SayHidden.s);

    }
}
```

Each module has a module-info.java file that describes module's dependencies. **Module1** contains two packages: **hi** and **hidden**. We want to expose only package **hi** to third-party modules and keep **hidden** private.

What we need to do is to define it in module-info.java file:

```java
module Module1 {
    exports hi;
}
```

We use the exports directive to expose all public members of the named package.

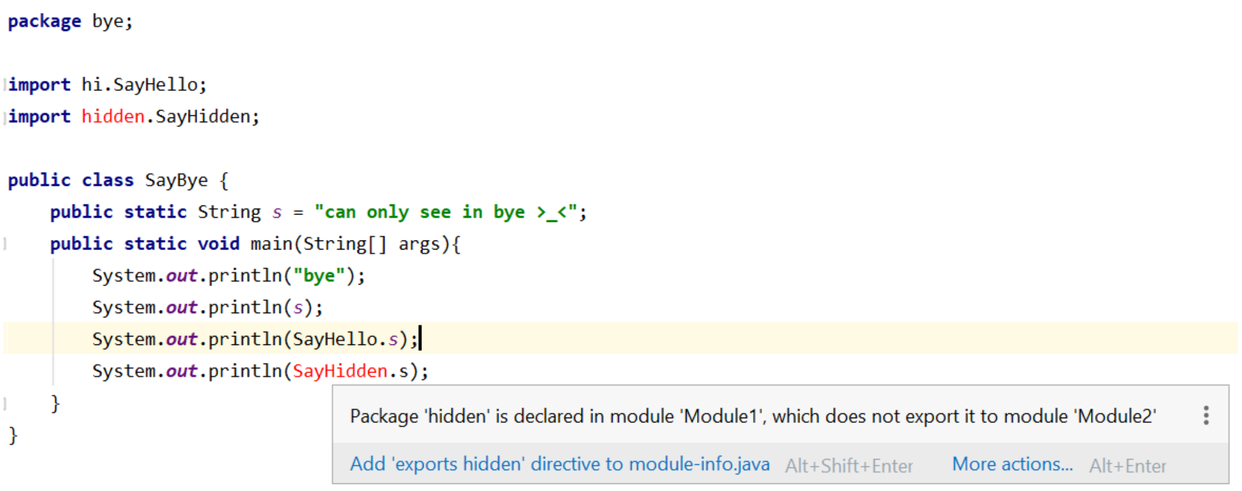
Class **SayBye** is defined in **Module2** and requires class **SayHello** defined in package **hi** of **Module1**, so we need to reflect it in module-info.java:

```java
module Module2 {
    requires Module1;
}
```

Note that in order for class **SayBye** to access the public variable **s** in **SayHello**, three conditions need to be fulfilled:

- Module1 needs to export package hi in its module-info.java
- Class SayBye need to import package hi
- Variable s need to be of public access

If you attempt to access variable s in SayHidden class, it will not be allowed if Module1 did not export package hidden.



```
package bye;

import hi.SayHello;
import hidden.SayHidden;

public class SayBye {
    public static String s = "can only see in bye >_<";
    public static void main(String[] args){
        System.out.println("bye");
        System.out.println(s);
        System.out.println(SayHello.s);
        System.out.println(SayHidden.s);
    }
}
```

Package 'hidden' is declared in module 'Module1', which does not export it to module 'Module2'

Add 'exports hidden' directive to module-info.java Alt+Shift+Enter
More actions... Alt+Enter

Note that Oracle has removed JavaFX from the Java Development Kit (JDK) 11, given an overall desire to pull out noncore modules from the JDK and retire them or stand them up as independent modules.

JavaFX 11, the first standalone release of the Java-based rich client technology, is downloadable at https://openjfx.io/. We will be using JavaFX 13 for this module.

When creating JavaFX program in JDK 13, it is essential to add JavaFX 13 jar files to the build path, and create the module-info.java file.

13.5 Creating a simple JavaFX Program using Pure Code

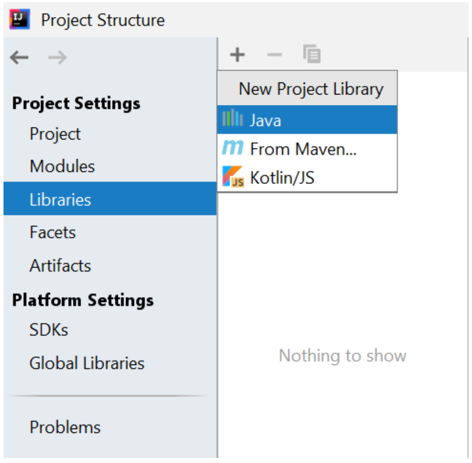
Now that we have a better understanding of Java's module structure, we can begin building our first JavaFX 13 program! Follow the steps below to create a simple JavaFX Program using pure code:

1. Create a new Java Project in IntelliJ. Give the project a name and click Finish.

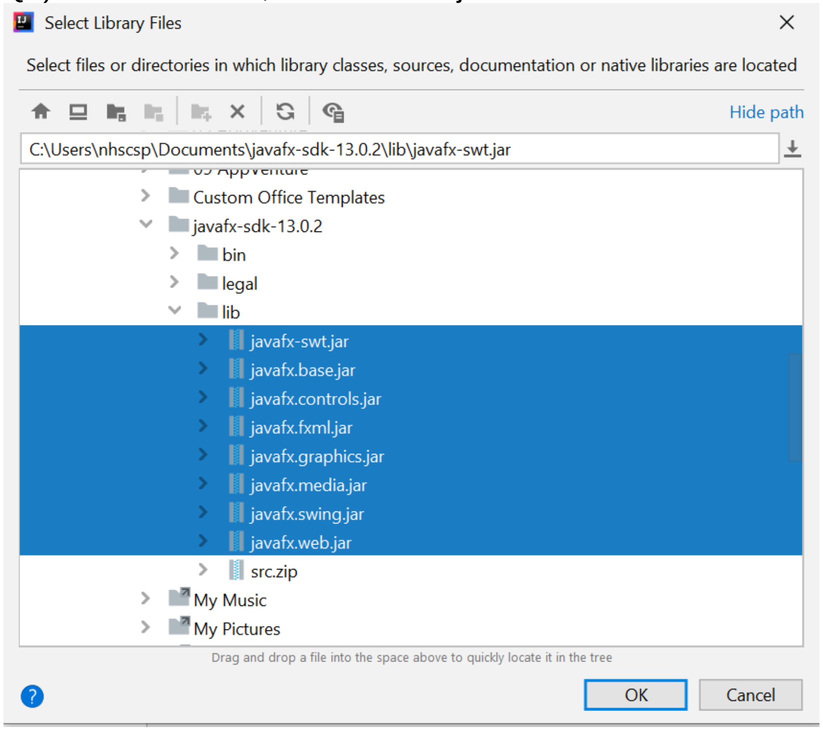
2. Add a new package. Name it **`sample`**.

3. Go to File > Project Structure. A window will pop up.

4. Under Project Settings, select Libraries.
   Click on the + icon and select Java.



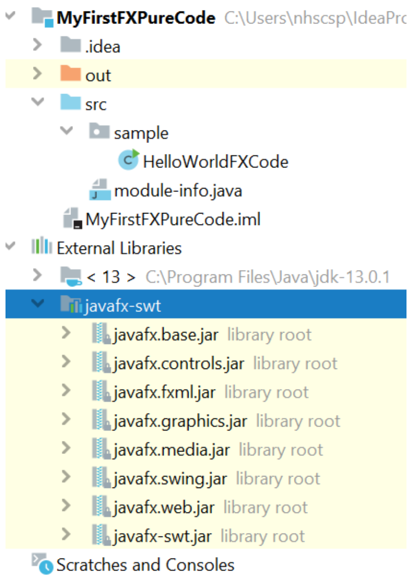
5. Navigate to the directory where you had saved the javafx-sdk-13 files (from Lab 13.0 Q1). In the lib folder, select all the .jar files and click OK.


Select Library Files

Select files or directories in which library classes, sources, documentation or native libraries are located

C:\Users\nhscsp\Documents\javafx-sdk-13.0.2\lib\javafx-swt.jar

6. Click OK > OK to go back to your project. The imported .jar files can now be found under External Libraries folder in your project.



```
MyFirstFXPureCode C:\Users\nhscsp\IdeaPrc
  .idea
  out
  src
    sample
      HelloWorldFXCode
    module-info.java
  MyFirstFXPureCode.iml
External Libraries
  < 13 > C:\Program Files\Java\jdk-13.0.1
  javafx-swt
    javafx.base.jar library root
    javafx.controls.jar library root
    javafx.fxml.jar library root
    javafx.graphics.jar library root
    javafx.media.jar library root
    javafx.swing.jar library root
    javafx.web.jar library root
    javafx-swt.jar library root
Scratches and Consoles
```

These files contains classes required to run a JavaFX program.

7. Right click on src folder.
   Select New > module-info.java.
   In module-info.java, paste the following code:

```
module MyFirstFXPureCode {
    requires javafx.controls;
    opens sample;
}
```

The module-info.java file specifies the following:

- The name of the module (i.e. `MyFirstFXPureCode`)
- The modules it depends on (i.e. `javafx.controls`)
- The package it opens (i.e. `sample`)

We need to opens package sample so that the JavaFX libraries can access and run the project.

If `opens sample` is not included, the following runtime error will be seen:

```
Exception in Application constructor
Exception in thread "main" java.lang.reflect.InvocationTargetException
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:567)
        at java.base/sun.launcher.LauncherHelper$FXHelper.main(LauncherHelper.java:1051)
Caused by: java.lang.RuntimeException: Unable to construct Application instance: class sample.HelloWorldFXCode
        at javafx.graphics/com.sun.javafx.application.LauncherImpl.launchApplication1(LauncherImpl.java:890)
        at
javafx.graphics/com.sun.javafx.application.LauncherImpl.lambda$launchApplication$2(LauncherImpl.java:195)
        at java.base/java.lang.Thread.run(Thread.java:830)
Caused by: java.lang.IllegalAccessException: class com.sun.javafx.application.LauncherImpl (in module
javafx.graphics) cannot access class sample.HelloWorldFXCode (in module MyFirstFXPureCode) because module
MyFirstFXPureCode does not export sample to module javafx.graphics
        at java.base/jdk.internal.reflect.Reflection.newIllegalAccessException(Reflection.java:376)
        at java.base/java.lang.reflect.AccessibleObject.checkAccess(AccessibleObject.java:642)
```

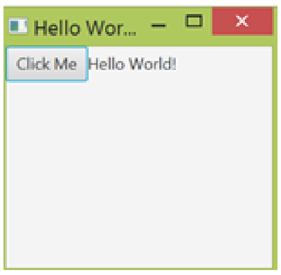
Required modules are not transitively available to transitive consumers, (ie: A requires B requires C means A does not see C automatically) unless requires transitive is specified. **Modules are required and packages are exported.**

8. Add a new Class named **`HelloWorldFXCode`** in the **`sample`** package with the code below:

```java
package sample;

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

//Application is the main class of a JavaFX program. Each JavaFX program must extend the
//Application class. This means that each FX program is a "child" of Application class.
//Thus all JavaFX program will inherit the attributes and methods from the Application class
public class HelloWorldFXCode extends Application {
   private Label label; //create a variable to reference a Label object
   private Button btn;  //create a variable to reference a Button object

    @Override //start() method is the main entry point of the application
    public void start(Stage primaryStage) {

        btn = new Button("Click Me");  //creates a button object
        label = new Label();           //creates a label object

        //connect btn to the code that runs when you click it
        btn.setOnAction(e -> buttonClick(e));

        FlowPane root = new FlowPane();//creates a pane to contain UI
        root.getChildren().add(btn);   //add the button to the pane
        root.getChildren().add(label); //add the label to the pane

        //create the scene object with specific dimensions. Add the pane into the scene
        Scene scene = new Scene(root, 300, 250);

        primaryStage.setScene(scene); //add the scene to the stage
        primaryStage.setTitle("Hello World!"); //set title of stage (i.e. window title bar)
        primaryStage.show();//show the stage (window)
    }
    //code to be triggered when user clicks on the button
    public void buttonClick(ActionEvent event) {
        System.out.println("You clicked me!");
        label.setText("Hello World!"); //change text of label. Note that you are able
                                       //to access label in this method as it's instance var
   }
   //main method is not necessary for JavaFX program.
   //It is only used as a fallback for situations in which JavaFX launching is not working.
    public static void main(String[] args) {
        launch(args);
    }
}
```

9. Compile and Run your program. Upon successful execution, you should see the following program. Clicking on the button will display the message "Hello World!".

   Congratulations!
   You have created your first JavaFX program!



One of the main advantage of coding the UI using pure Java code is you will be able to quickly create many similar UI using loops! The program below demonstrates how can you quickly generate 20 Label objects, and update the text to a random integer with each button click.

```java
package sample;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;
import java.util.ArrayList;
import java.util.Random;

public class MultipleLabels extends Application {

    private ArrayList<Label> labelList;

    @Override
    public void start(Stage primaryStage) {
        labelList = new ArrayList<>(); //create an ArrayList to store the Label objects.

        FlowPane root = new FlowPane();
        Button btn = new Button("Randomize");
        Random r = new Random();

        //use a loop to quickly generate 20 Label objects
        for(int i = 0; i < 20; i++){
            //create a new Label object in each iteration.
            //Set the text to a random int in range [0,49]
            Label l = new Label(r.nextInt(50)+"");
            l.setPrefSize(100,50); //set size of Label to be 100 by 50
            l.setStyle("-fx-border-color: black"); //set the border
            l.setAlignment(Pos.CENTER); //set alignment to center

            //add Label object l to arraylist so that you can quickly access them using loops
            //in the event handler.
            labelList.add(l);
            root.getChildren().add(labelList.get(i));//add each Label object to the Flow pane.
        }

        root.getChildren().add(btn);
        btn.setOnAction(e -> buttonClick(e));

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

        primaryStage.setScene(scene);
        primaryStage.setTitle("Number Randomizer");
        primaryStage.show();

    }

    public void buttonClick(ActionEvent event) {
        Random r = new Random();
        //retrieve each label object from the arraylist of Labels.
        //For each Label, randomize a new integer value and set the label text.
        for(int i = 0; i < labelList.size(); i++){
            labelList.get(i).setText(r.nextInt(50)+"");
        }
    }
}
```

13.6 JavaFX Stage Object

In JavaFX, a Stage object is a window. A Stage object called primary stage is automatically created by the JVM when the application is launched.

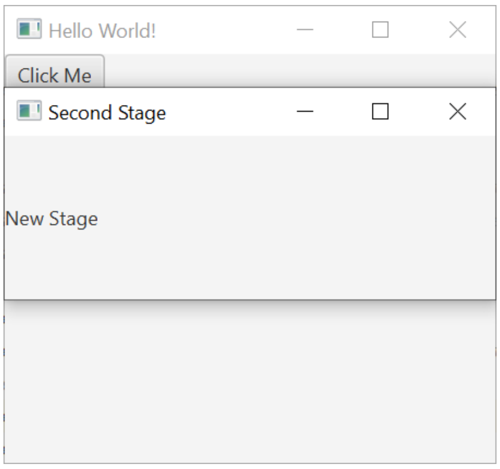
`primaryStage.setScene(scene)` sets the scene to the primary stage and `primaryStage.show()` displays the primary stage. JavaFX names the Stage and Scene classes using the analogy from the theater. You may think stage as the platform to support scenes and nodes as actors to perform in the scenes.

You can create additional stages if needed.

For example, you may insert the following code in the start() method to add another stage.

```
Stage stage = new Stage(); // Create a new stage
stage.setTitle("Second Stage"); // Set the stage title
// Set a scene with a button in the stage
stage.setScene(new Scene(new Label("New Stage"), 400, 100));
stage.show(); // Display the stage
```

The following will be seen when the program is executed. The stage will be displayed on top of each other, but can be closed and moved around independently.



Below summarizes some other useful methods pertaining to a stage.

<u>Stage position</u>
You can set the position (X,Y) of a JavaFX Stage via its setX() and setY() methods. The setX() and setY() methods set the position of the upper left corner of the window represented by the Stage. For example:

```
stage.setX(50);
stage.setY(50);
```

<u>Stage Height and Width</u>

```
stage.setWidth(600);
stage.setHeight(300);
```

Stage Modality

You can set window modality of a JavaFX Stage. The Stage modality determines if the window representing the Stage will block other windows opened by the same JavaFX application. You set the window modality of a JavaFX Stage via its **`initModality()`** method.

For example:

```
Stage stage = new Stage();
//choose one of the below modality
stage.initModality(Modality.APPLICATION_MODAL);
stage.initModality(Modality.WINDOW_MODAL);
stage.initModality(Modality.NONE);
```

There are 3 types of Modality:

- **Modality.APPLICATION_MODAL** the stage will block all other windows (stages) opened by this JavaFX application. You cannot access any other windows until this Stage window has been closed.

- **Modality.WINDOW_MODAL** the newly created Stage will block the Stage window that "owns" the newly created Stage, but only that.

- **Modality.NONE**: the stage will not block any other windows opened in this application.

The Modality.APPLICATION_MODAL and Modality.WINDOW_MODAL modality modes are useful for Stage objects representing windows that function as "dialogs" which should block the application or window until the dialog process is completed by the user (for example a pop up to warn user of an error). The Modality.NONE modality is useful for Stage objects representing windows that can co-exist, like different browser windows in a browser application.

Stage Owner

A JavaFX Stage can be owned by another Stage. You set the owner of a Stage via its **`initOwner()`** method.

For example:

```
Stage stage = new Stage();
stage.initModality(Modality.WINDOW_MODAL);
stage.initOwner(primaryStage);
```

This example will open a new Stage which will block the Stage owning the newly created Stage (which is set to the primary stage).

Stage Full Screen Mode

You can switch a JavaFX Stage into full screen mode via the Stage **`setFullScreen()`** method. Please note, that you may not get the expected result (a window in full screen mode) unless you set a Scene on the Stage.

```
primaryStage.setFullScreen(true);
```

For details on how to code other UI controls, you may wish to explore further via this tutorial:
http://tutorials.jenkov.com/javafx/index.html