

2020 CI401
Introduction to programming

Week 1.10
Introduction to JavaFX

Dr Roger Evans
Module leader
8th December 2020

Lecture recording and attendance

- This lecture will be recorded and published in the module area
- The focus of recording is on the lecturer, not the audience
- If you are particularly concerned not to be part of the recording, turn off your microphone and camera.
- In addition, lecture attendance is now being routinely recorded (in all modules) to help the School Office monitor engagement
- (This slide is really a reminder to me to start recording and record attendance!)

Module structure (version 3)

Semester 1

Week	Topic	Theme
1.01	Introduction / Hello World	Coding
1.02	Variables, loops and choices	Coding
1.03	Input, more loops and choices	Coding
1.04	Variables and expressions	Coding
1.05	Types, assignment and arrays	Data
1.06	Let's play Top Trumps!	Data
1.07	Objects and methods	OO
1.08	Working with numbers	Data
1.09	Simple Algorithms	Dvp
1.10	Introduction to JavaFX	Dvp
1.11	Simple Animation	Dvp
	Xmas vacation 21 Dec – 8 Jan	
1.12	GUIs using MVC	OO
1.13		

Semester 2

Week	Topic	Theme	Project
2.01	Project topics and assessment	Project	Set
2.02	Simple Inheritance	OO	Lab
2.03	Scope, Visibility and Encapsulation	OO	Lab
2.04	Testing - JUnit	Testing	Lab
2.05	Documentation - Javadoc	Doc	Study
2.06	Collections and generic types	Data	Study
2.07	IO: files and streams	Dvp	Study
	Easter Vacation 29 Mar – 16 Apr		
2.08	Numbers – the computer's view	Data	Study
2.09	Java vs Python		Submit?
2.10	More algorithms – search and sort	Dvp	
2.11	How fast is my code?	Dvp	
2.12	Java 'under the hood'		
2.13	Revision week		Exam ↓

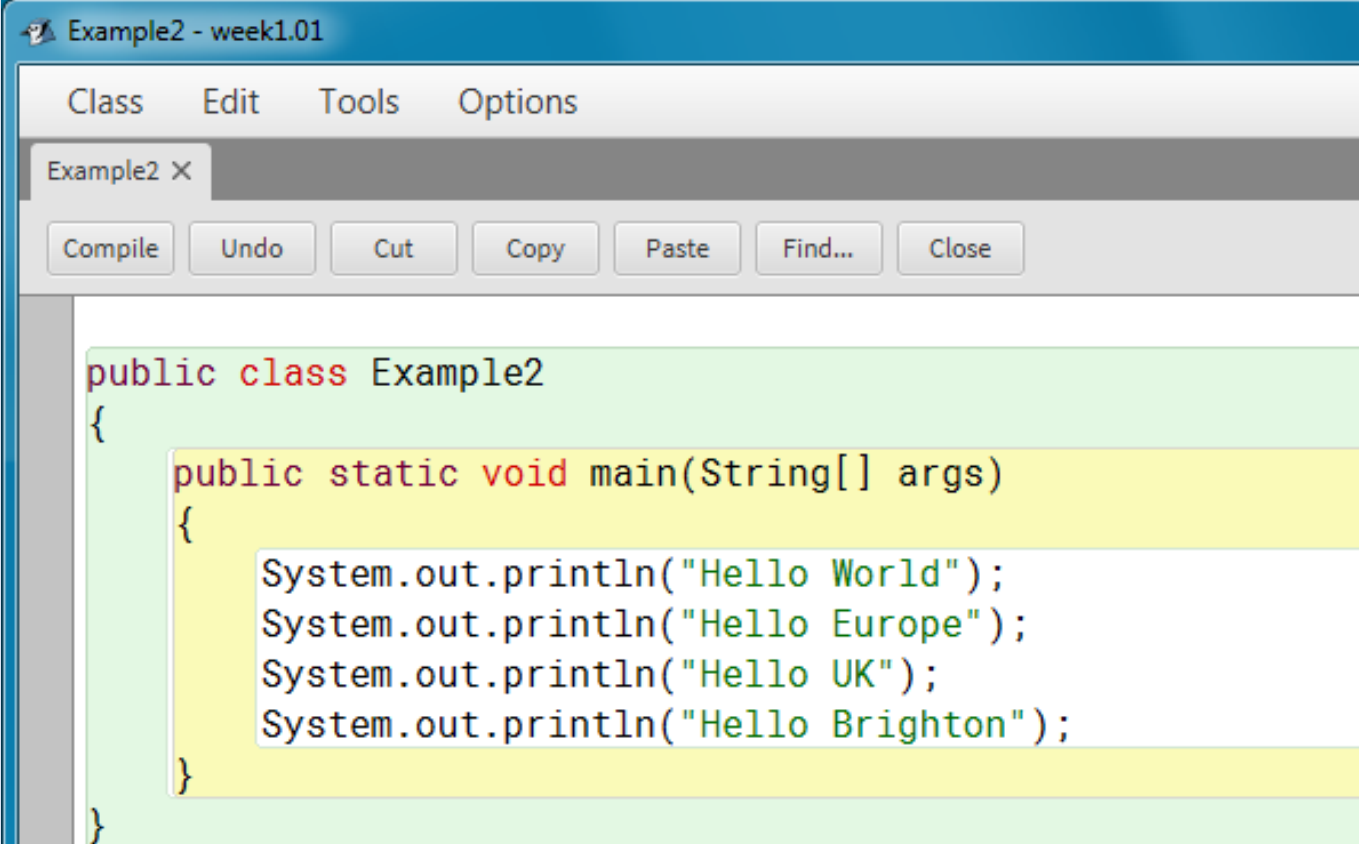
A 'new' approach to programming

Our first model of programming

Single 'main' method

Simple sequence of commands for the computer to execute

Mostly just printing things



```
public class Example2
{
    public static void main(String[] args)
    {
        System.out.println("Hello World");
        System.out.println("Hello Europe");
        System.out.println("Hello UK");
        System.out.println("Hello Brighton");
    }
}
```

The screenshot shows a Java IDE window titled "Example2 - week1.01". The window has a menu bar with "Class", "Edit", "Tools", and "Options". Below the menu bar is a toolbar with buttons for "Compile", "Undo", "Cut", "Copy", "Paste", "Find...", and "Close". The main editing area contains the following Java code:

Our first model of programming

Even with more complex instructions – loops and if statements

We are still basically giving the computer a complete list of things to do, in a specific order

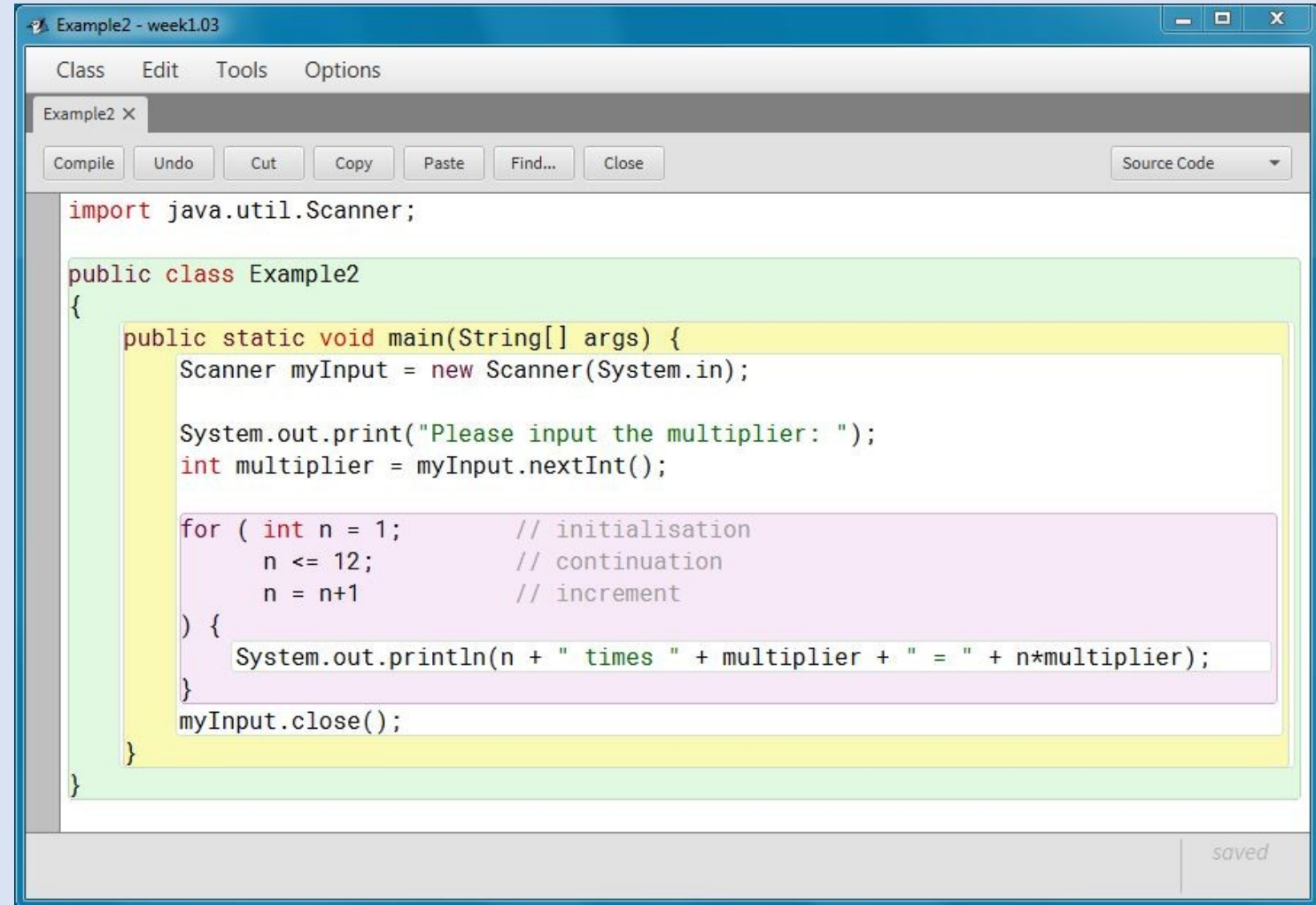
And still mostly just printing things

```
// a very simple estate-agent bot. Give it a list of rooms
// to visit (as 'command-line' arguments) and it will generate
// a commentary. See what happens if you visit a room twice
public class Example3
{
    public static void main(String[] args)
    {
        String lastRoom = "";
        String lastButOneRoom = "";
        for (String room: args)
        {
            // using 'print' instead of 'println' to stop each
            // part printing on a different line
            System.out.print("Here we are ");
            //see whether we have just come back here
            if (room.equals(lastButOneRoom)) {
                System.out.print("back ");
            }
            System.out.print("in the " + room + ". ");
            if (! room.equals(lastButOneRoom)) {
                // comment about individual rooms.
                if (room.equals("kitchen")) {
                    System.out.print("Can you smell the coffee? ");
                } else if (room.equals("bedroom")) {
                    System.out.print("Plenty of room for all your clothes.");
                } else if (room.equals("lounge")) {
                    System.out.print("Notice the original fireplace.");
                }
            }
            System.out.println();
            lastButOneRoom = lastRoom;
            lastRoom = room;
        }
    }
}
```

Adding user input

We added `Scanner`, which allowed the user to contribute to the process

Our program simply stopped and waited for the user to type something (we say the program **blocked**, waiting for input)



```
import java.util.Scanner;

public class Example2
{
    public static void main(String[] args) {
        Scanner myInput = new Scanner(System.in);

        System.out.print("Please input the multiplier: ");
        int multiplier = myInput.nextInt();

        for ( int n = 1;          // initialisation
              n <= 12;           // continuation
              n = n+1            // increment
            ) {
            System.out.println(n + " times " + multiplier + " = " + n*multiplier);
        }
        myInput.close();
    }
}
```

saved

'Traditional' programming

- A single sequence of instructions (sometimes known as a **thread**)
- Text-based input and output – using the **console** device
- **Synchronous, blocking** input (and output) – the program stops and waits for the user to type something before going on

'Traditional' programming (2)

- But we all know that the computers around us don't work like that!
- They are:
 - Doing many things at the same time
 - Accepting non-text inputs (such as mouse gestures, clicks, touch screens, voice and video) and producing non-text output
 - Carrying on doing things whether we are inputting data or not
- Or at least they seem to be ...

Non-text IO – WIMP interfaces

- Around 40 years ago (!), a new development in user interface completely changed the way users interacted with computers
- This was the invention of Windows, Icons, Menus, Pointer (WIMP) interfaces
- And direct manipulation interfaces – where you have a visual view of your computer world that you can change directly (instead of typing instructions):
 - Deleting a file old style: type `$ del myfile.doc` into a command shell
 - Deleting a file by direct manipulation: drag the file icon into the wastebasket

Supporting WIMP interfaces

- Suddenly users were multi-tasking, doing several different things on one computer, typing partial input into one, switching to another task etc.
- And they also expected the systems to remain responsive – not freeze, or fail to update their screens properly, (that is not to **block** on input)
- This approach was supported by two new software technologies:
 - Event-driven programming
 - Graphical user interfaces

Event-driven programming

Event-driven programming

- In **event-driven programming**, instead of writing programs which run in a sequential way, you write individual pieces of code which respond to **events** occurring in the computer 'world', such as a user action, a system issue, or a program-generated event.
- So instead of writing a program which says '**now ask the user for some input**' (and wait for it), you write code for '**what to do when the user types some input**' – an **input event**
- The code written to respond to an event is called an **event handler**.

Event-driven programming

- A program is written as a set of 'event handler' methods, each of which is designed to respond to a particular event
- Events might be
 - user input (key presses)
 - actions external to the program such as a window becoming visible on the screen, or a filename changing
 - actions generated by the program itself (eg update game state)
- Multiple event handlers can respond to the same event (eg filename change in File Explorer) – they don't always need to know about each other.

Event-driven programming – how it works

- Event-driven programs look different from traditional programs
- In particular they do not seem to have a **main** method – the code you write appears to never get called!
- What's happening is that there is a 'secret' main method, called the **event loop**
- The event loop just sits there waiting for an event to happen and when it does, it call all the event handlers that are interested in that event.
- The event loop keeps doing this until one of the event handlers terminates the program.

Event-driven programming

- Traditional programming
 - Program specifies exact sequence of instructions from start to finish
 - When input is required, program stops and waits for user to type/do something
 - Program then continues running
- Issues
 - Doesn't fit an object-oriented code structure very well
 - Does not cope well with multi-tasking (eg GUI interfaces)
 - Everything stops while waiting for input
- Event-driven programming
 - No complete specification from start to finish
 - Program specifies how to respond to 'events' (such as, but not only, user input)
 - Appears not to 'do' anything
 - (actually it is running an 'event-loop' in secret)
 - Fits multi-tasking well – *as long as response to events are quick enough*
 - Fits o-o structure well – state and behaviour managed in small chunks (object)

Graphical User Interfaces (GUIs)

Graphical user interfaces

- Graphical user interfaces (or GUIs) are software systems which provide the familiar style of interface we see on modern computers
- These typically support multiple **windows** on the screen, often laid out in a layered overlapping configuration, which are movable, resizable, can be minimized/maximized etc.
- A simple program runs completely within a single window, but more complex programs can have multiple windows.

GUIS – controls and layout managers

- Within one window, a single coherent user interface is typically provided using common features such as buttons, type-in boxes, menus, scrollbars etc..
- These elements are known as **controls**, and are linked to underlying data and application logic.
- Controls are laid out and managed by **layout managers**, which are responsible for sizing and arranging controls, and resizing them if the window resizes

GUIs and event-driven programming

- GUIs work with **event-driven programming** to create an application
- **Controls** and **layout managers** display their user interface on the screen, and then wait for user events.
- When an event happens, a **control** will run an **event handler**, which may change what it shows on the screen, and/or update some other part of the program
- Then it will finish, and the program will wait for the next event

JavaFX – a GUI library for Java

JavaFX

- **JavaFX** is a new(ish) library package to support the needs of modern GUI development in Java
- It is an alternative to **Swing**, which has been the main Java GUI library for over 20 years
- It provides support for multiple platforms, integrates with modern technologies such as CSS, and HTML5, and has a visual layout tool, Scene Builder, for the development of GUIs without coding
- (In CI401, we are developing GUIs **with** coding ...)

Our first JavaFX program

- As always, let's start with **Hello World**
- But in a GUI program, we don't write Hello World to the console, we display it in our own GUI window

Hello World

- Lots of 'imports' (don't worry about them, just include them!!)
- A **main** method (this starts the secret loop, though we don't actually use it in BlueJ)
- Variables for a **GridPane** layout manager and a **Label** control (to hold the message)
- A **start** method – this is where we make the layout manager and control, make a **Scene**, add to the **window** and display it

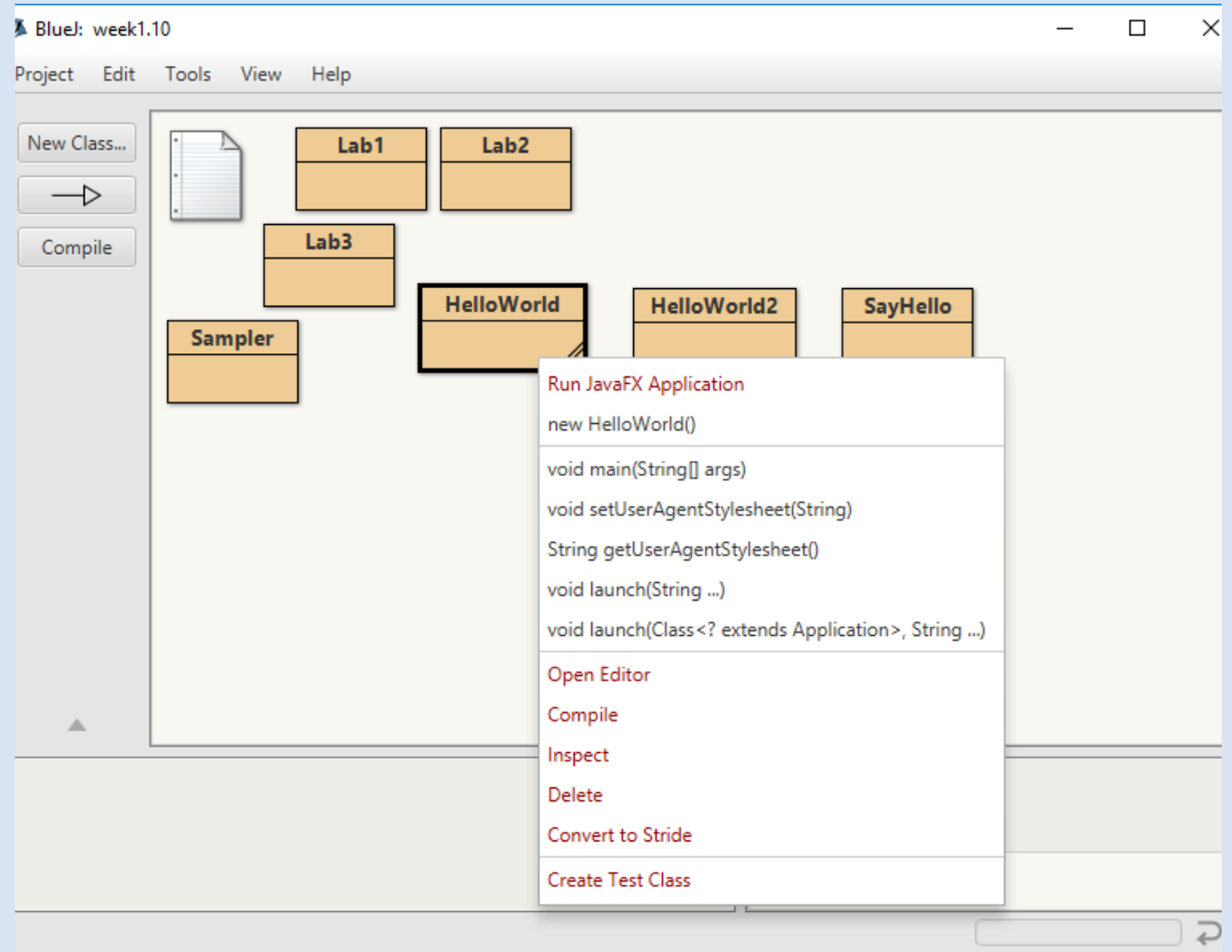
```
1 import javafx.application.Application;
2 import javafx.event.ActionEvent;
3 import javafx.event.EventHandler;
4 import javafx.scene.Scene;
5 import javafx.scene.control.*;
6 import javafx.scene.layout.*;
7 import javafx.scene.text.*;
8 import javafx.stage.Stage;
9
10 // A Hello World program using the JavaFX graphical user interface library
11 public class HelloWorld extends Application {
12     // This 'main' method allow us to run Hello World as a free-standing Java program
13     public static void main(String[] args) {
14         launch(args);
15     }
16     // a GridPane object to lay out our graphical interface
17     public GridPane grid;
18     // a Label object to convey our message on the screen
19     public Label label1;
20
21     // this method gets called to start up our program
22     // it gets passed a Stage object which represents the window on the screen
23     public void start(Stage window) {
24         // Set the title on the window
25         window.setTitle("Hello World Example");
26         // Make a layout manager
27         grid = new GridPane();
28         // Make a Label object saying Hello World message and add it to
29         // the grid in position 0,0
30         label1 = new Label("Hello world!");
31         grid.add(label1,0,0);
32         // make a new Scene from the grid, add it to the window and display it
33         window.setScene(new Scene(grid, 300, 250));
34         window.show();
35     }
36 }
```


Running HelloWorld in BlueJ

Compile the class, and then right-click on it

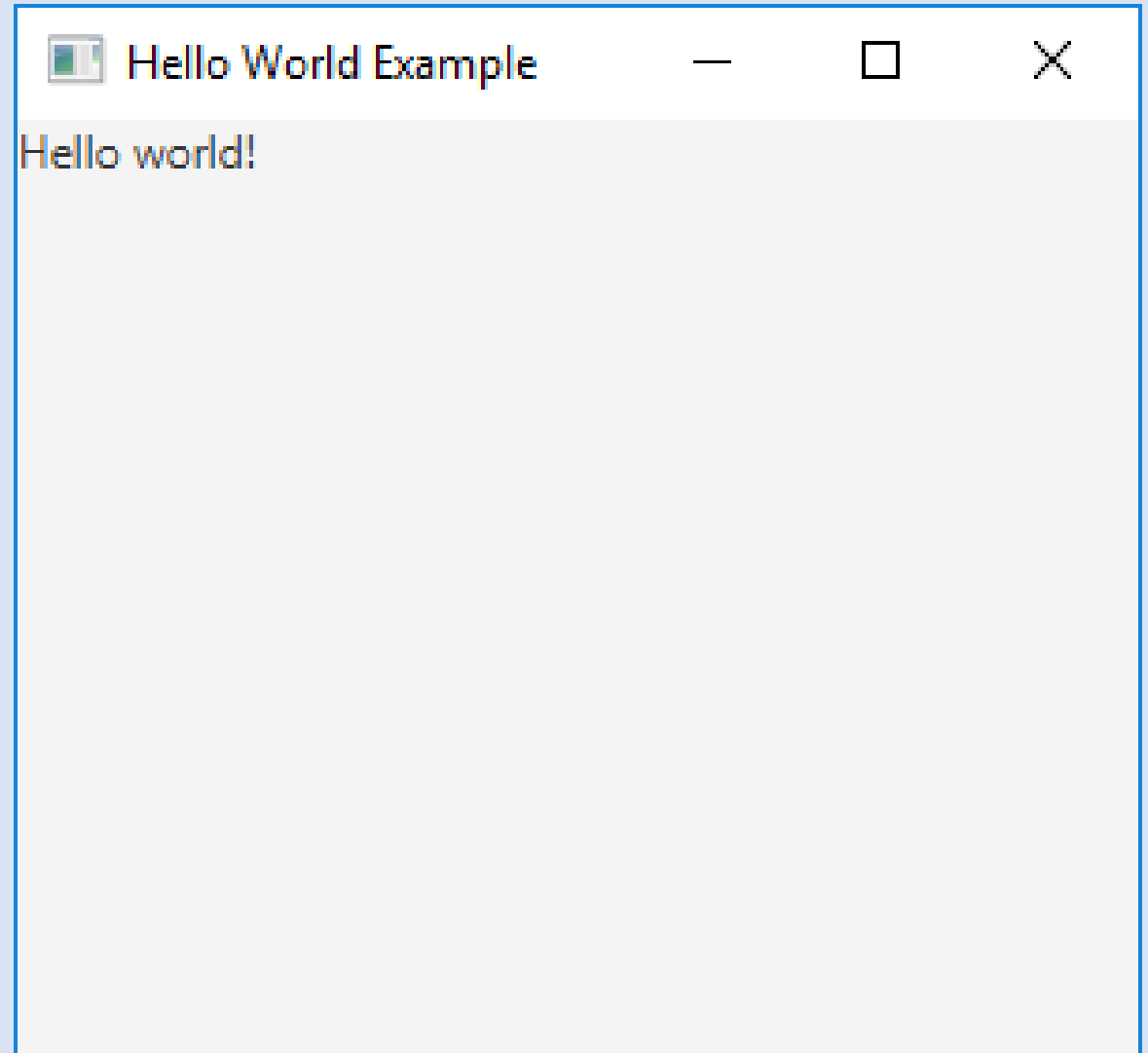
The menu that appears will have a new option 'Run JavaFX Application'

Select this option to run the code



HelloWorld output

The program produces a new window, with 'Hello World!' displayed in it



So what happened?

- When you clicked **Run JavaFX Application**, BlueJ made an instance of the **HelloWorld** class, and called its **start** method
- The argument to the **start** method was a **Stage** object called **window**, created by BlueJ, which is like a bit of real estate on the actual computer screen for the program to use
- The **start** method made two objects
 - A **GridPane**, which is a layout manager, which lays out other controls in a grid pattern
 - A **Label**, which is just a box on the screen to hold a **String**
- The String inside the label was set to '**Hello World**'
- The Label was added to the GridPane, and then the GridPane was attached to the window (via a **Scene** object, which doesn't matter too much, but it set the size of the final window – 300 by 250)
- Finally, the window was told to display its contents

What didn't happen

- We never ran the **main** method – this is because BlueJ is already running the secret event loop itself. The **main** is there in case you want to run this program outside of BlueJ. It will make a **Stage** object, call **start** and then run the secret event loop.
- We didn't add any **event handlers**, so this program doesn't really do anything, except display **Hello World!**
- So let's be a little more exciting

HelloWorld2

This has all the bits **HelloWorld** had,
plus a **Button** object (a control) called
button1

label1 is initially set to 'Wait for it ...'

When **button1** is clicked, **label1**
changes to 'Hello World!'

```
// HelloWorld2 adds a little bit of interactivity to our example
// The 'Hello World' message is not displayed until the user clicks
// a button
public class HelloWorld2 extends Application
{
    // Main method for freestanding use (not BlueJ)
    public static void main(String[] args) {
        launch(args);
    }

    // the layout object
    public GridPane grid;
    // the GUI objects - a label as before and a button
    public Label label1;
    public Button button1;

    public void start(Stage window) {
        window.setTitle("Hello World 2 Example");
        // make the grid
        grid = new GridPane();

        // make the label and position it on the grid
        label1 = new Label("wait for it ...");
        grid.add(label1,0,0);

        // make the button, with its own label ('Click here'), and position it on
        // the grid below the label
        button1 = new Button("Click here");
        grid.add(button1,0,1);
        // this line tells the system to run the method 'button1Click' (defined
        // below) when the button is clicked
        button1.setOnAction(this::button1Click);

        // set the window Scene and show it on the user's screen
        window.setScene(new Scene(grid, 300, 250));
        window.show();
    }

    // this method runs when the button is clicked
    // it changes the text in the label object to be 'Hello world!'
    public void button1Click(ActionEvent event) {
        label1.setText("Hello world!");
    }
}
```

HelloWorld2

We make **button1** and add it to the layout manager, below **label1**

The button has a label '**Click here**'

The button doesn't do anything, but it registers an event handler:

setonClick(this::button1Click);

This is telling JavaFX that if the button is clicked, it should run the method **button1Click**

button1Click changes **label1** to say 'Hello World'

```
public void start(Stage window) {  
    window.setTitle("Hello World 2 Example");  
    // make the grid  
    grid = new GridPane();  
  
    // make the label and position it on the grid  
    label1 = new Label("wait for it ...");  
    grid.add(label1,0,0);  
  
    // make the button, with its own label ('Click here'), and position it on  
    // the grid below the label  
    button1 = new Button("Click here");  
    grid.add(button1,0,1);  
    // this line tells the system to run the method 'button1Click' (defined  
    // below) when the button is clicked  
    button1.setOnAction(this::button1Click);  
  
    // set the window Scene and show it on the user's screen  
    window.setScene(new Scene(grid, 300, 250));  
    window.show();  
}  
  
// this method runs when the button is clicked  
// it changes the text in the label object to be 'Hello world!'  
public void button1Click(ActionEvent event) {  
    label1.setText("Hello world!");  
}  
}
```

SayHello

- SayHello is similar, but has a Text box (that you can type into)
- When you type your name, and hit <enter>, the program says hello to you.
- These are very simple examples of how you use managers, controls and event handlers to create an event-driven GUI application in JavaFX

Anatomy of SayHello

Declarations

- A layout object
- Three controls

Create objects

Add event handler

Create Scene content and add to scene

The event handler

- Gets text from `textField1` and modifies text in `label1`

```
18 public class SayHello extends Application
19 {
20     // Main method for freestanding use (not BlueJ)
21     public static void main(String[] args) {
22         launch(args);
23     }
24     // the layout object
25     public GridPane grid;
26     // The GUI objects - two labels and a text field
27     public Label label1;
28     public Label label2;
29     public TextField textField1;
30
31     public void start(Stage window) {
32         window.setTitle("Say Hello");
33         grid = new GridPane();
34
35         // make label1 - where our message will appear
36         label1 = new Label("wait for it ...");
37         grid.add(label1,0,0);
38         // label2 tells the user what the text field is for
39         label2 = new Label("Name: ");
40         grid.add(label2,0,1);
41         // we put the textfield next to label2 (in a second column on the grid)
42         textField1 = new TextField();
43         grid.add(textField1,1,1);
44         // set the method to call (when the user presses <enter> in the text field)
45         textField1.setOnAction(this::textField1Click);
46
47         // add the content and show the window
48         window.setScene(new Scene(grid, 300, 250));
49         window.show();
50     }
51
52     // Event handler - what to do when the user presses <enter> in the text field
53     public void textField1Click(ActionEvent event) {
54         // get the text the user typed from textField1
55         String userText = textField1.getText();
56         // set label1, but this time using the user-supplied text
57         label1.setText("Hello " + userText);
58     }
59 }
```


Key parts of a JavaFX app

- **Stage** object which generally maps onto a window on the screen.
- **Scene** object which represents the view/management of the GUI in the window
- A **layout manager**, such as a **GridPane**, which holds the content of the GUI (individual controls or other layout managers).
- **Controls**, such as **Labels**, **Buttons**, **Textfields** etc. which provide main user interface functions
- **Event handlers**, methods which allow the app to 'do something' when a user event happens (eg a button press)

07548626529

Working with JavaFX

JavaFX, BlueJ and Eclipse

- JavaFX is a standard Java library, so much of the code development etc., is completely standard and supported by standard tools.
- However, because JavaFX provides GUI support, there are some points at which it may interact or interfere with the GUI support of the tools used to develop systems, such as **BlueJ** or **Eclipse**

JavaFX and BlueJ – 3 main points

1. When creating a new BlueJ class, you should select the class type **JavaFX Class**, rather than just **Class**. This loads a convenient template for JavaFX classes, and also allows BlueJ to handle them differently when running a class.
2. A class which has been created as a JavaFX Class has a new option on the class menu: **Run JavaFX Application**. This is the option to use to test a class you have created.
3. JavaFX application classes are created with a 'main' method. This method allows your code to run as a standalone Java program. However, if you try and run the main method from BlueJ (as you would any other Java program class), it will generate an error. Instead use option 2.

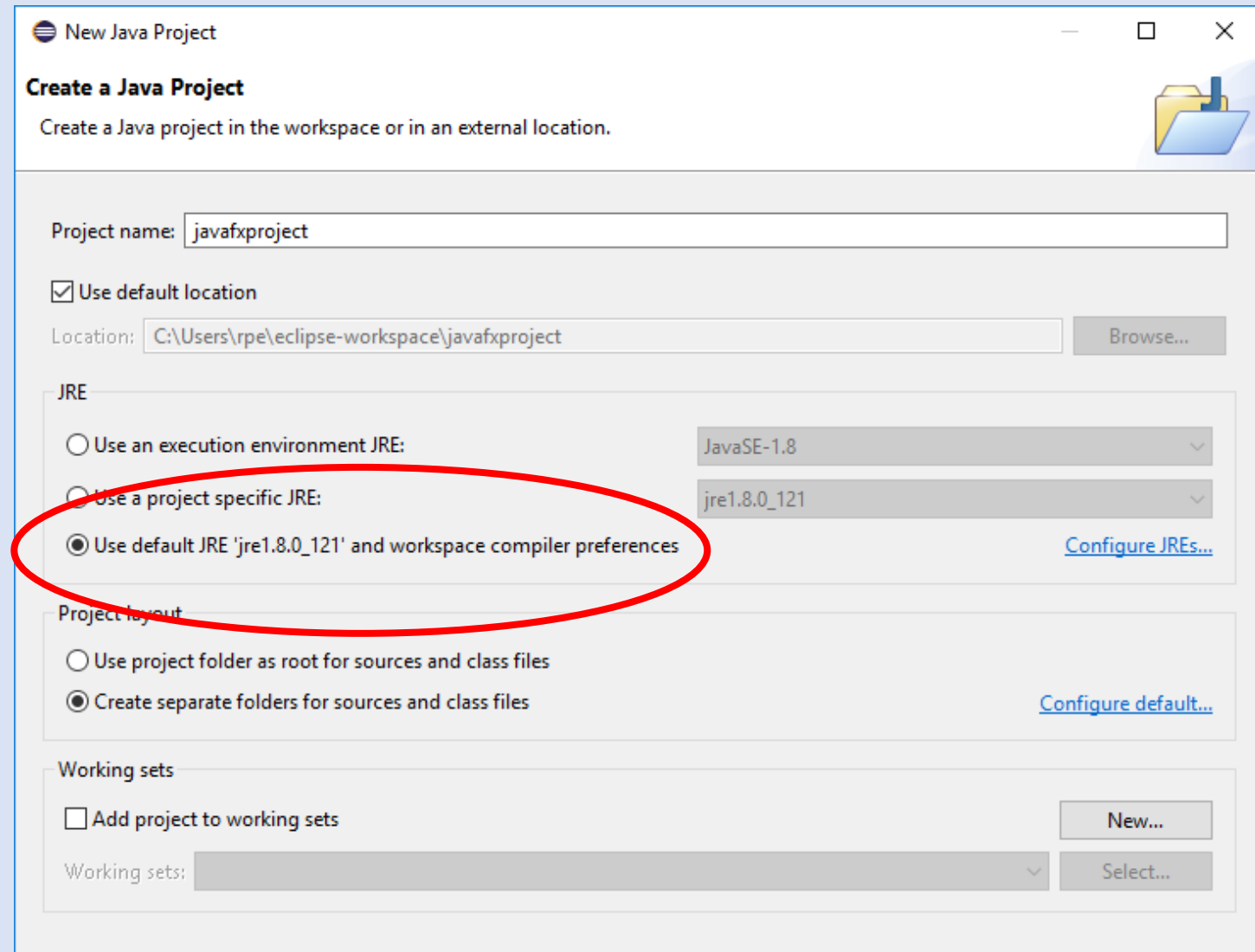
JavaFX and Eclipse

- The standard installation of Eclipse for Java does not automatically include the JavaFX libraries in new Java projects. This means your JavaFX classes **will not compile**.
- To create a project which does include JavaFX, you need to change the **JRE** setting when you create the project – see next slide for details
- If you have your own eclipse installation, you might like to install the **e(fx)clipse** plugin (<https://www.eclipse.org/efxclipse/index.html>), which provides many more JavaFX-related tools. I've not played with this yet though, so you are on your own with it (though I'm always happy to try and help).

Creating a JavaFX project in Eclipse

To create an Eclipse project which does include JavaFX, select the **third** JRE option ('Use default JRE')

(A standard Java project uses the first option, which does not include JavaFX)



Lab exercises

Week 1.10

Week 1.10 labs

- HelloWorld, HelloWorld2 and SayHello are the examples from the lecture
- Lab1 – extend SayHello to be an event-driven version of the EstateAgent app (or the QueenBot)
- ControlsDemo shows you how to use some other input controls
- Lab2 extends ControlsDemo into a ‘report form’

Getting help

- **If you get stuck, or get behind, ask for help in labs!**
- **Code-clinic sessions (team code o727xhp):**
 - Mondays-Wednesdays: 1pm to 2pm.
 - Thursdays: 12pm to 1pm.
 - Fridays: 10am to 11am.