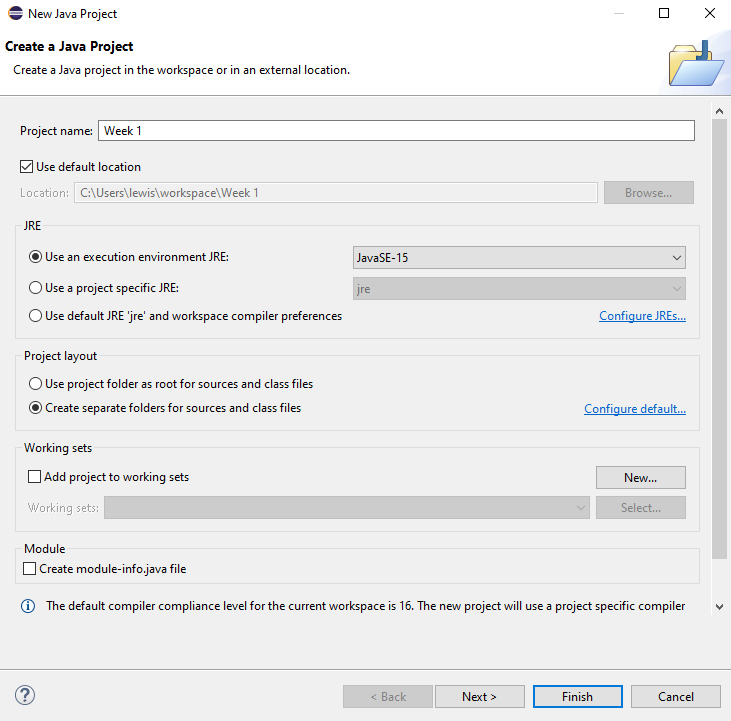
Week 1, Lab A – Console Applications

# Lab Intro & Prep

This lab will serve as an introduction to using a more industry-relevant Integrated Development Environment (IDE) known as Eclipse. We will develop simple console-based applications, using a combination of techniques learned in Programming 1 and in this week’s lecture.

## Learning Objectives

* Become familiar with Eclipse
* Develop simple console-based applications
* Obtain user input via the Scanner class
* Begin by launching **Eclipse**
* If asked to select your workspace, make sure you select the **W: drive** (a dedicated drive set up for your workspace) – **Do NOT store your workspace on OneDrive (causes syncing issues)**
* Create a new Project by selecting **New > Java Project.**
* Name the project **Week 1** (Fig 1) – this will serve as your workspace (folder) for all of the programs you make during this week. Untick **Create module-info.java file** (last checkbox at the bottom). Click **Finish.**

****

**Untick** ‘Create module-info.java file’ if you see this option

**Name your project ‘**Week1’

Fig - New Java Project Window

# Exercise 1 – Simple Console Output

One of the first programs you create when learning a language is a simple ‘Hello World’ application, which simply prints “Hello, World!” to the console. Let’s test our environment works by making such a program.

1. In your **Week 1** project folder, create a Class (**File > New > Class**) named **HelloWorld** (Fig 2) that contains a **main method** (tick the public static void(String[] args) checkbox).

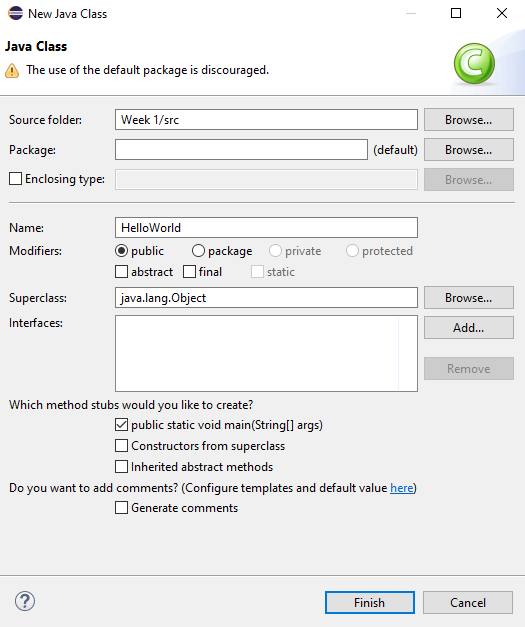


Fig - New Class Window

1. Enter print/println statements to print “Hello, World!” to the console (Fig 3)

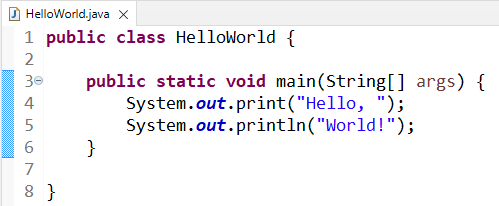


Fig - HelloWorld.java

1. Run the program by pressing the green play button to the right of the bug icon (Fig 4) to test your environment works

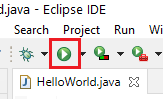


Fig - Run Button

# Exercise 2 – Introducing Variables

1. Introduce three new variables to your program to record your **name**, **age** (in whole years), and **height** (in meters). Think about what data types would be most suitable for these variables.
2. Enter individual println statements to print your **name**, **age**, and **height** to produce the output below (Fig 5).

**Hints:**

* A **String** is a data type that allows for a sequence of characters to be stored
* Any floating-point value typed directly into your source code (e.g. float height = 1.8;) will be treated as a **double** (a floating-point value with double precision). Attempting to store a double into a float data type will lead to an error: **Type mismatch: cannot convert from double to float.** This can be solved by casting the double to a float, or by appending the value with an **f** – e.g. 1.8f (tells the compiler to store the value as a float).

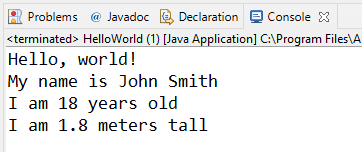


Fig – Exercise 2 Example Output

# Exercise 3 – User Input

Using the **Scanner** class (introduced in the lecture), create a class named **Averages** with a main method that allows the user to enter 5 integer numbers and display the total and average of the numbers, both as floating-point values (Fig 6).

**Note:** We will improve this program in Part B of the lab to include input validation (prevent the user giving us other *types* of input).

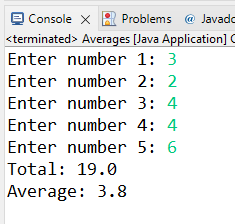


Fig – Exercise 3 Example Output

# Exercise 4 – Table of Squares and Cubes

Create a new class named **SquareTable** with a main method. Print to the console, in a table-like format, the first twelve positive integers, along with the result of the numbers being squared and cubed (Fig 7).

**Hint**: Instead of using spaces, a \t can be included in a string will print a tab, allowing the columns to be evenly distributed. This is known as an escape sequence.

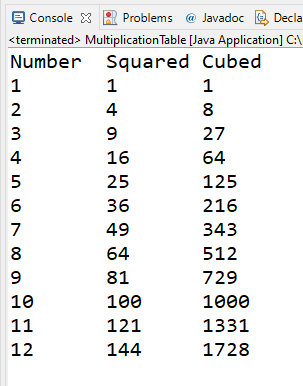


Fig – Exercise 4 Example Output

# Extension Exercise 1 – BankTeller

This exercise will involve developing a Bank Teller console application that will calculate the optimum combination of notes to be handed out to customers (e.g. if they want to withdraw cash from the bank). Create a new class called **BankTeller** with a main method.

A variable **sterling** contains the numbers of pounds sterling to be handed out. The program should then work out how many **fifties** (£50 notes), **twenties**, **tens**, **fives, twos** and how many **one**-pound coins should be issued (Fig 8). For example, an optimum combination of notes/coins for a sterling amount of £131 would be 2x £50 notes, 1x £20 note, 1x £10 note, and 1x £1 coin. Starter code is provided in Listing 1.

**Hint:** the modulus operator (%) gives the remainder after division, e.g.:

5%2 == 1 11%3 == 2

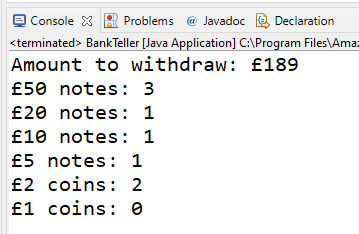


Fig – BankTeller Example Output

|  |
| --- |
| **public** **class** BankTeller {  **public** **static** **void** main(String[] args) {  **int** sterling = 189;  **int** fifties, twenties, tens, fives, twos, ones;  }  } |

Listing – BankTeller Starter Code

# Extension Exercise 2 – BankTeller (with pence)

Amend your **BankTeller** application to include pence. Your sterling variable will need to change to a floating-point data type. You will also need to introduce new variables to record the number of 50p, 20p, 10p, 5p, 2p and 1p coins to be handed to the customer.

**Design Ideas:**

* Split pounds from pence – cast to / truncate or use modulus
* Multiply pence by 100
* Deal with pounds
* Deal with pence