**Week 8 Lab – Intro to Console Applications**

# Lab Intro

This will be our first lab using a different **Integrated Development Environment** (IDE), IntelliJ. **IntelliJ** is just one of many IDEs that allows us to program using the Java syntax and is a **more industry relevant IDE used by professional Java developers**.

There will be quite a lot of exercises in this lab, giving you an opportunity to practise a lot of the concepts we’ve seen so far in the unit, but using the IntelliJ environment. Remember that **Programming is a 30-credit unit**, which **assumes 300 hours of study** (this includes times spent in lectures, labs, and webinars). It’s important you put the hours in to ensure the rest of your degree runs smoothly.

## Learning Outcomes

* **Develop** simple console-based applications
* **Develop** interactive programs that allow user input
* Store and process a collection of primitive types (using an array)
* Store and process a collection of object types (using an ArrayList)

## Resources

* Week 7 Lecture Slides (these introduced the array and ArrayList collections)
* Week 8 Lecture Slides (these introduced console applications and IntelliJ)

# Preparation

Start by launching IntelliJ from the **Start** menu. When it launches, you should be greeted with a screen like the one below (Figure 1).

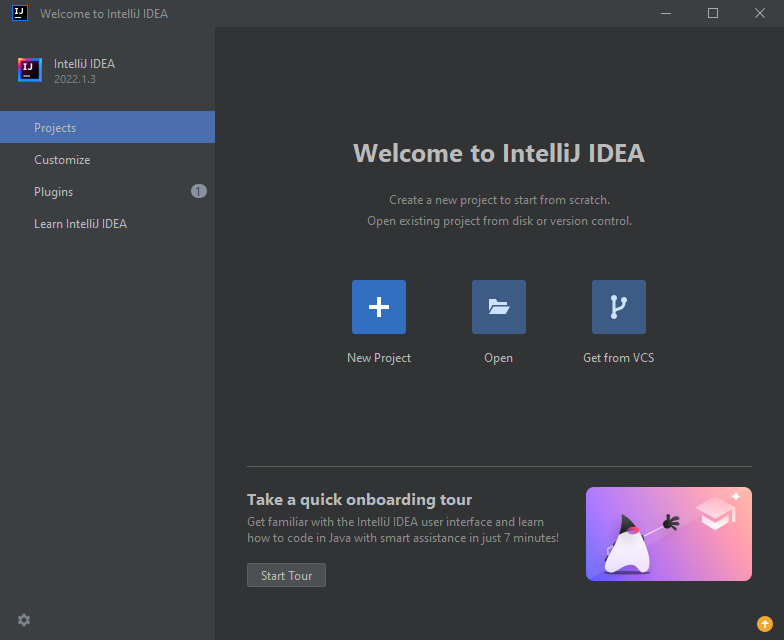


Figure IntelliJ Launch Screen

If you want to change the colour theme of IntelliJ, now is a good time to do so. You can do this by clicking **Customize** (Figure 2) and selecting one of the four built-in themes that come with **IntelliJ**. I will be using IntelliJ Light from this point onwards, but feel free to select the one best for you. You can also change the default font size on this screen if 12 is too small.

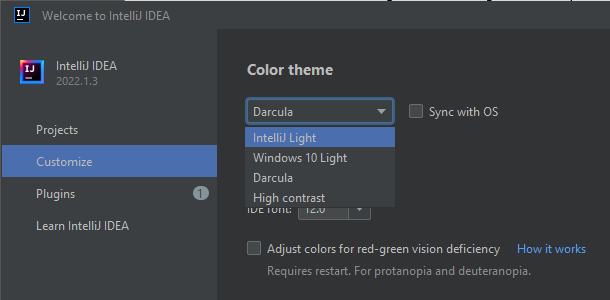


Figure Customize IntelliJ Theme

Navigate back to the **Projects** tab (Figure 3). This tab allows you to launch a project in one of three ways: (1) create a new project (2) open an existing project, or (3) Open a project from a Version Control System (e.g. Git). Select **New Project** to start a new IntelliJ project.

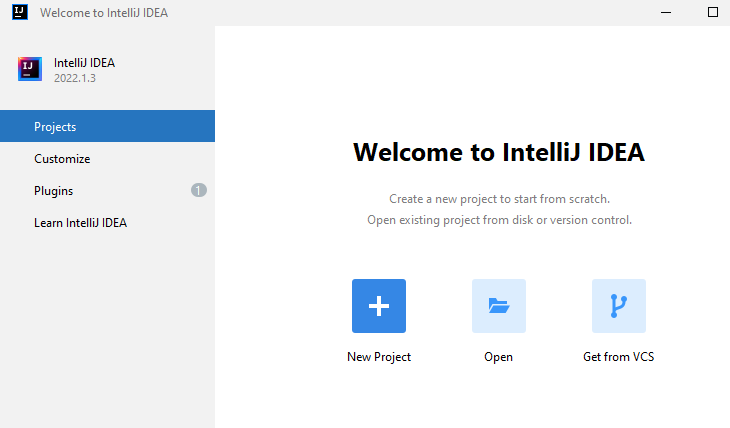
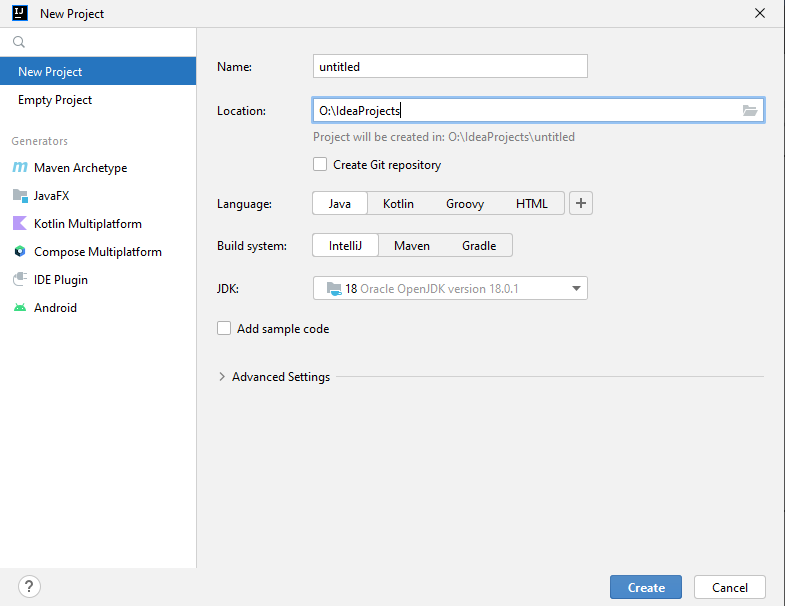


Figure Projects IntelliJ Tab

You should be greeted with the screen below (Figure 4) – see the annotated notes for what the various fields mean.

**Name** of the Project – spaces are allowed. A project will be represented as a folder in your file system.



**Location** of the Project – where your project will be saved

**Java Version** – we will be using the default – JDK 18 (JDK = Java Development Kit, allows us to write and compile Java applications)

Figure IntelliJ Project Creation Screen

See Table 1 for what values you should be using when creating your new project.

Table IntelliJ New Project Fields

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Value to Use** |
| Name | This will be the name of your project. When you create the project a folder will be created that matches the project name | Week 8 |
| Location | Where your project (folder) will be stored.  **DO NOT USE THE O:\ DRIVE!!**  The O:\ drive represents your OneDrive folder. Although OneDrive can handle syncing individual Processing (.PDE) files perfectly file, it will struggle to sync an IntelliJ project, as the project folder will contain many smaller files.  We will be using the W:\ drive – this is a storage area available to you when on a campus machine. If you want to continue to work on your exercises outside of the lab, the bottom of this lab sheet (Appendix A) shows how to export your project as a ZIP file, which can then be imported into IntelliJ on another machine. | W:\  **Note:** you may want to create a subfolder in W:\ named **Programming** and set that as your location. This will then create a folder for your project at **W:\Programming\Week 8** |
| Language | The language our project will use. Keep this default. | *Leave the default selected (Java)* |
| Build System | The build system. Keep this default. | *Leave the default selected (IntelliJ)* |
| JDK | The version of the Java Development Kit (JDK) to use for this project. We will keep the default. | *Leave the default selected (18)* |

# Exercise 1 – Simple Console App



Let’s start off by creating a console application that simply prints something to the console to ensure our environment works.

1. **Create** a **Class** (right-click the **src** folder and select **New** -> **Java Class**) – see Figure 5.
2. Name the Class **HelloWorld** (Figure 6)
3. In this class, add a **main method** – you can either type out the main method if you remember the syntax from the lecture, or simply type “main” into the class bracers { } and IntelliJ should recommend adding a main method from there (Figure 7).

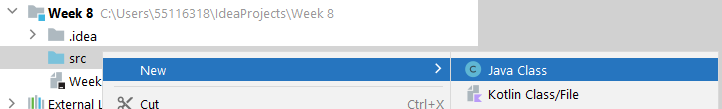


Figure Creating a New Class in IntelliJ

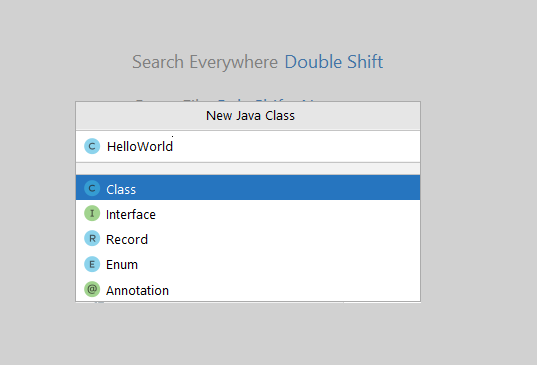
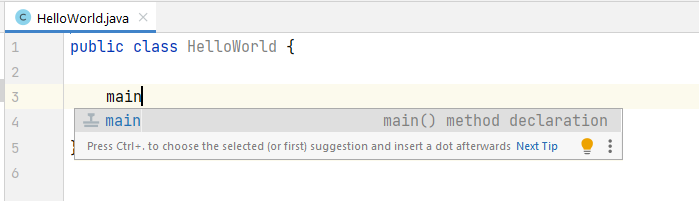


Figure Naming a Class in IntelliJ



**Key Term – main method**The main method begins execution of a Java program.



Figure IntelliJ Suggestion

1. Enter a println statement in the main method that prints “Hello, world!” to the console (recall from the lecture that printing to the console can be achieved using System.out.println). **Hint:** entering **sout** brings up IntelliJ’s code suggestion for printing a string to a console (Figure 8).
   * System.out.print() – prints something to the console and leaves the cursor on the current line
   * System.out.println() – prints something to the console and moves the cursor to the next line

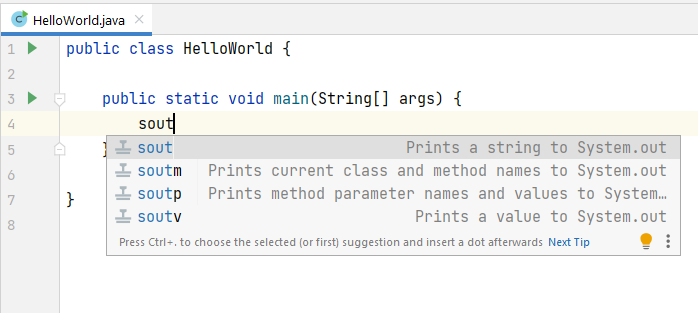


Figure sout shortcut for generating a println

1. **Run the application** – there are many ways you can run the program, the green “play” button at the top-right is probably the easiest (Figure 9)

Graphical user interface, text, application, email

Description automatically generated

Figure IntelliJ's Run Button

You should see “Hello, world!” output to the console area at the bottom (Figure 10).

Graphical user interface, text, application, email

Description automatically generated

Figure Exercise 1 Console Output

# Exercise 2 – Introducing Variables



1. In your **HelloWorld** class, introduce three new variables in the main method to record your **name**, **age** (in whole years), and **height** (in meters). Think about what data types would be most suitable for these variables.

**Key Term – Scanner**The **Scanner** class is a built-in Java class that allows us to gather input from the keyboard.



1. Enter individual println statements to print your **name**, **age**, and **height** to produce the output below (Figure 11). Use the variables created within the println statements using string concatenation (+ operator)

**Hints:**

* A **String** is a data type that allows for a sequence of characters to be stored
  + String variableName = “ ”;
* 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).

Graphical user interface, text, application

Description automatically generated

Figure Exercise 2 Example Output

# Exercise 3 – Calculating Average of User Input



1. Create a new Class named **Averages**.

**Key Term – escape sequence**Escape sequences signal an alternative interpretation of a series of characters. In Java, a character preceded by a backslash (\) is an escape sequence. **\n** = newline **\t** prints a tab – see online for others!



1. Add a **main method** to the class.
2. Using the **Scanner class** introduced in the lecture, allow the user to enter 5 integer numbers and display the total and average numbers as floating-point values. The output should resemble that of Figure 12.

**Hints:**

* Remember that the **Scanner** class needs to be imported into our program to be able to use it. You can either add the import declaration manually at the top of the code outside of the class ( **import java.util.Scanner;** ), or tell IntelliJ to import it by moving your mouse over Scanner and clicking on Import class (Figure 13).

Graphical user interface, text, application

Description automatically generated

Figure 12 Exercise 3 Example Output

Graphical user interface, text, application

Description automatically generated

Figure Importing a Class



# Exercise 4 – Table of Squares and Cubes

1. Create a new class named **SquareTable** with a main method.
2. Print to the console, in a table-like format, the first twelve positive integers, along with the result of those numbers being **squared** and **cubed** (see Figure 14 for what your output should look like). **Note:** you should use a for loop and do the calculations, not just simply print out the numbers “manually”.

**Hints:**

* You don’t need a **Scanner** for this exercise, since we’re not getting any user input
* Instead of using spaces, a **\t** can be included within a string will print a tab, allowing the columns to be evenly distributed. This is known as an escape sequence.

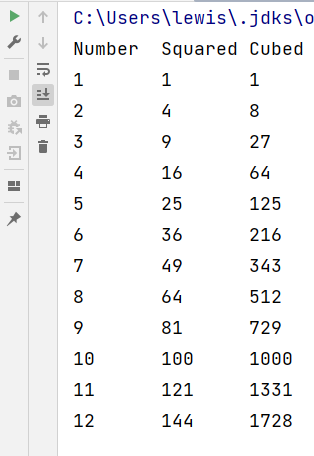


Figure Exercise 4 Example Output

# Exercise 5 – Fahrenheit to Celsius Converter



This exercise will involve fetching a double from the user that represents a temperature in Fahrenheit, and will convert it into Celsius. Figure 15 shows an example output of this program.

1. Create a class named **TempConverter** and add a main method to it
2. Prompt the user to enter a temperature in Fahrenheit as a double (store this input in a double variable)
3. **Store** the user input in a **double** variable named **fahrenheit**
4. Convert the Fahrenheit value entered into Celsius (search for the formula online and see if you can implement it into code)
5. Print out the Celsius value to the console

Text

Description automatically generated

Figure Exercise 5 Example Output



# Exercise 6 – Array of ints

This exercise will involve storing an array of integers, where the size of the array is entered by the user via a Scanner. You will then loop through the number of elements of that array and ask the user for each individual element to be stored, and then loop through the array to print out all of the even numbers. The pseudocode/steps for this are provided below, with an example output of this program provided in Figure 16.

1. Create a new Class with an appropriate name (e.g. ArrayEvenNumbers) and add a main method to it.
2. Create a Scanner object so that we can collect user input.
3. Using a System.out.print, ask the user how many numbers they will be entering .
4. Store the user’s input in an integer variable.
5. Create an array of ints, using the input you’ve just stored in the integer variable in the previous step.
6. Using a traditional for loop (i.e. a for loop that uses an i counter variable), loop through the int array you’ve created, prompting the user for each value (you will need a for loop and this for loop will contain a print statement to ask the user for the current element/value, and then store it in the relevant index.
7. Using a for-each loop, loop through the array and print out the element **only if the element is an even number** (**hint:** the modulus operator % gives the remainder after a division).

**Note:** we don’t really need this additional for loop, since we could of course calculate the total as the user is entering them, but it gives us a chance to separate the logic and use a for-each to iterate over the array.

Text, table

Description automatically generated with medium confidence

Figure Exercise 6 Example Output

# Exercise 7 – ArrayList of Doubles



We’ve seen a few examples of ArrayLists now, and all of those examples have stored objects of classes we’ve defined – e.g. an ArrayList of Aliens in Lecture 7, and an ArrayList of Birds in the last couple of webinar sessions.

We also know there are **eight primitive types** in the Java language, highlighted pink in Figure 17. Primitive types have one job – to store a value of its type (e.g. an int can store a whole number, with no fractional part allowed). They are not like objects, since we cannot call methods off of a primitive int variable. E.g if we have an int variable called **number**, we can’t say something like **number.render()**, since int is a primitive – and primitives don’t have methods.

Can we have an ArrayList that stores **ints** or an ArrayList that stores **doubles**? Technically no, ArrayList can only store objects – or more specifically, references to those objects.

**Key Term – Wrapper class**A **Wrapper Class** is the class version of a primitive. All of the eight primitive types in Java have a corresponding class version.



There is a workaround to this however – every single one of the eight primitives has a class equivalent (e.g. the class version of a **double** is the **Double** class). Table 2 lists the eight primitive types and their class equivalents. These class versions of primitives are referred to as **Wrapper Classes**, since they “wrap” around their smaller primitive counterparts.

Diagram

Description automatically generated

Figure Java's Eight Primitive Types

Table Class Versions of Java's Eight Primitive Types

|  |  |
| --- | --- |
| **Primitive Type** | **Class Type** |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |

This means if we want an ArrayList of doubles, we can use declare the data type the ArrayList will hold as **Double, like so:**

ArrayList<Double> temperatures = new ArrayList<>();

For this exercise, we are going to create an **ArrayList of doubles** to allow a user to continuously type in temperatures, and then calculate the average of the temperatures that were typed in. Since the person isn’t going to tell us how many temperatures they are going to give us like the previous exercise, we can use an ArrayList so they can continuously give us input as the program is running.

When they’re finished giving us temperatures, they can give us a value which signals that they’ve finished entering input – this is referred to as a **sentinel value** – a value that is not treated as part of the dataset, but is instead used to signal end of data entry.

1. Create a class called **AverageTemperature** with a main method
2. Create a scanner object, since we’ll be collecting some input
3. Create an ArrayList of doubles (remember to use the class version of the double)
4. Replicate the functionality that is present in

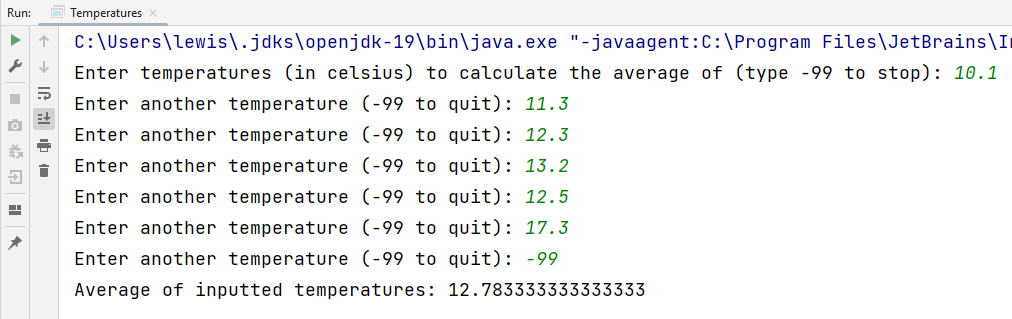


Figure Exercise 7 Example Output



# Extension Exercise – 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).

A variable **sterling** should contain the numbers of pounds sterling to be handed out as a whole number (int). The program should then work out how many **fifties** (£50 notes), **twenties**, **tens**, **fives, twos** and how many **one**-pound coins should be issued

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.

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

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

# Appendix A – Exporting an IntelliJ Project

If you want to continue working on these exercises outside of the lab, this section will quickly show you how you can export your project as a ZIP file.

Select your Project in the Project Explorer and select **File** -> **Export** -> **Export to Zip File…** (Figure 19)

Graphical user interface, text, application

Description automatically generated

Figure Export Project to Zip File

When selecting a location for this ZIP file, you can select your OneDrive (since OneDrive will find it much easier to sync a single ZIP file than it will trying to sync an IntelliJ project with lots of small files in it).

If you then want to open this project, you simply unzip the ZIP file and go to **File -> Open…** and browse for the extracted folder.

# Appendix B – Disabling “Reassigned local variable”

If the grey underline on local variables is annoying you, you can disable it by going to **File > Settings…** and navigating to the screen seen inFigure 20**.**

Graphical user interface, text, application

Description automatically generated

Uncheck

Figure Option for Disabling "Reassigned local variable" message