**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Practical 1 – Structured Programming Review**

Welcome back for year 2! You have already covered so much of the Java basics in first year, and hopefully the knowledge you gained will stand to you as you progress through this module. The main purpose of this first lab sheet is simply to reacquaint you with the **syntax** of Java and to get the brain **solving problems** again. We all get rusty over the summer break, so the lab sessions this week will just involve exercises based on the first year Java modules. It will also introduce the IDE we will be using for this module, **IntelliJ**.

As always with programming, the main thrust of the **course centers around** **you** - you will obtain **hands-on experience** of designing solutions and then writing, debugging and testing code using the programming language Java**,** while I will guide you along the way.

As it is a **10-credit** module, there will be a large amount of material to cover, so do **engage** right from the start, **work hard** to make up any Java deficiencies you might have carried from 1st year and make sure you put in **extra time at home** whenever necessary, to make sure you don’t fall behind.

**Course Outline**

• Principles of OOP

• Fundamental OO Concepts

• Event-handling and API GUI Classes

• Code Conventions

* Documenting OO Systems
* Data Structures and Persistence

**Exams**

It is expected that you will be given **2 continuous assessments** during this module, **accounting for 50% of the total marks** for the module. Both assessments will be practical in nature, where you will have the computer and a Java IDE/compiler available to you. The first CA will be an in-class exam **worth 20%** (around week 6) and the second CA will be a take-home mini-project, **worth 30%**, that you will work on for about three weeks. The second CA will commence around the 13th November or so.

The **final exam** in December will be **worth 50%** and note that this will be a **practical exam** (with possibly some theory questions thrown in also), testing your knowledge of the entire module. You need to get at least 40% overall to pass the module. Try to attend all your labs - **attendance** becomes a factor where students are “**borderline cases**” and a good attendance record will stand in your favour.

**What do You Get from Successful Completion of this Module?**

• The ability to explain the fundamental OO concepts and principles.

• The ability to formulate problems as steps to be solved systematically and implement the solutions programmatically making use of classes from the standard API.

• The ability to design, write and test small OO systems making use of user-defined classes, using inheritance and interfaces at a basic level, and adhering to standard coding conventions on style and security.

• The ability to use data structures from the collections framework and implement modules that write such structures to or read them back from persistent storage, dealing with any exceptions that may arise.

• The ability to use UML diagrams, an automatic documentation tool such as Javadoc, and version control, to develop and document a system effectively.

**Getting Started**

Some of you may have transferred from other courses and some may be Erasmus students, who have not taken the 1st year “Structured Programming” modules. In any case, many people will simply be rusty on their Java after the long summer break and also there are those who really struggled with Java in first year. Therefore, the first 2 lab sheets are designed to very quickly review the material covered in year 1. Especially if you are a newcomer, **consult the first year Java lab sheets** for more detail. I will give all newcomers access to these lab sheets on X: drive, let me know if you require access to them.

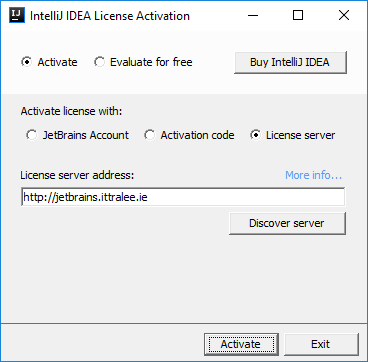
**Logging on to the PC**

The machine will boot to Windows 10 by default. Hit return and click OK to accept the **Computer Services Policies**. Now the login box appears. If available, type in your student t-number and password when prompted. Click **OK** to log on. You should now be logged in to the Windows 10 operating system and the college network.

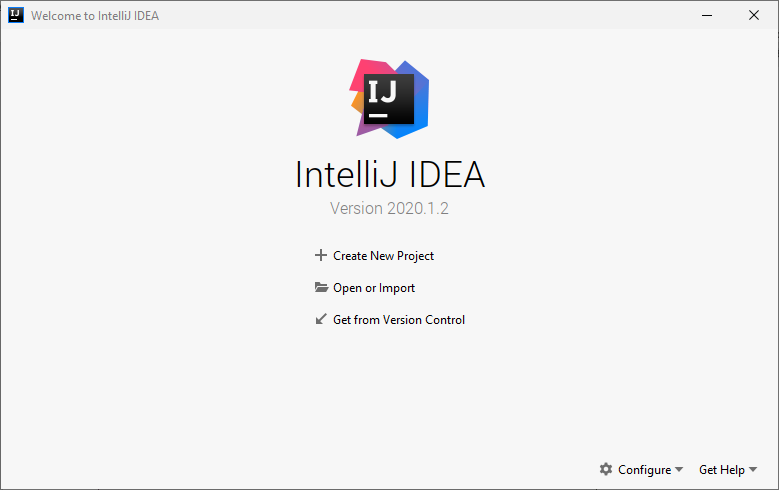
**Getting into IntelliJ**

Click on the **Search Windows** button on the taskbar (it looks like a magnifying glass) and type in the letters “in” - hopefully you will get a match for **IntelliJ IDEA**.

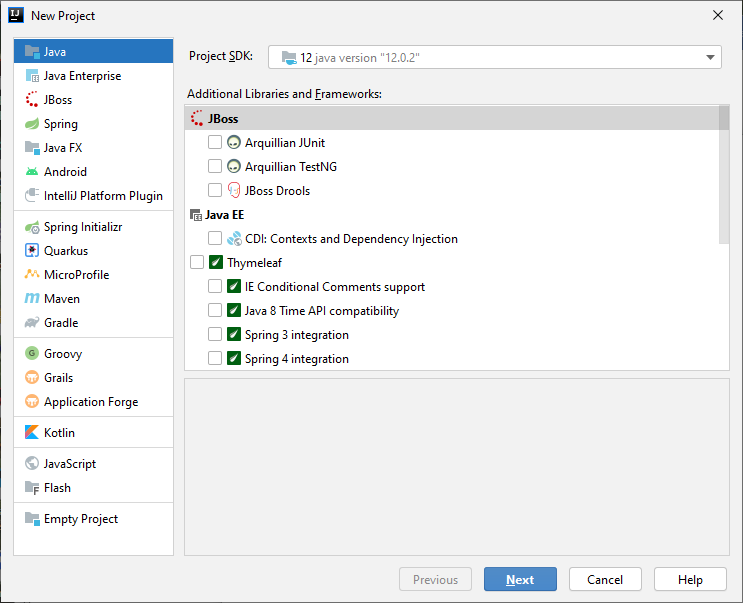
You will see the screen below. Simply select **License Server** 🡪 **Discover Server** 🡪 **Activate**



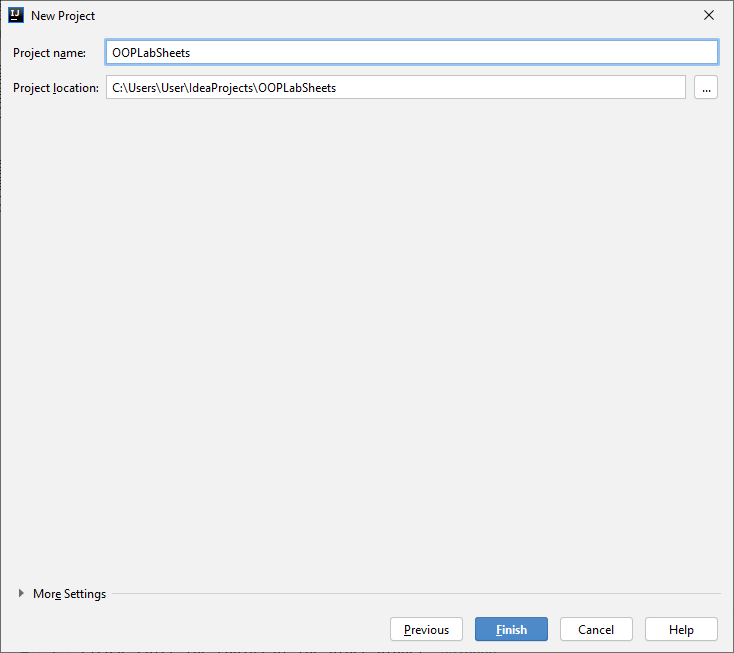
When completed, the IntelliJ **Integrated Development Environment** (**IDE**) should launch for you, after a few seconds. Once this happens, you are ready for coding!



To begin with above, you will see options to “Create New Project”, “Open or Import” etc. Click “**Create New Project**” and then the following window appears.

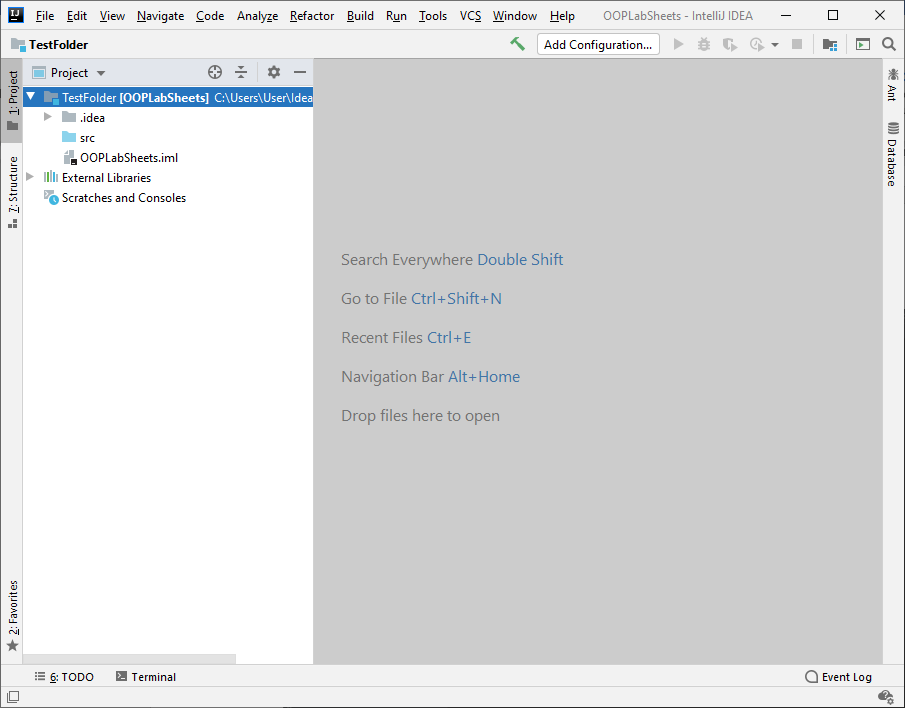


On this window you will select “**Java**” as the type of project and choose the appropriate “**Project SDK**” (in our case this will be some version of **Java 12**). Click **Next**, and then **Next** again to the screen that follows. Then you will be presented with a screen that allows you to pick a name for your new project and its location.

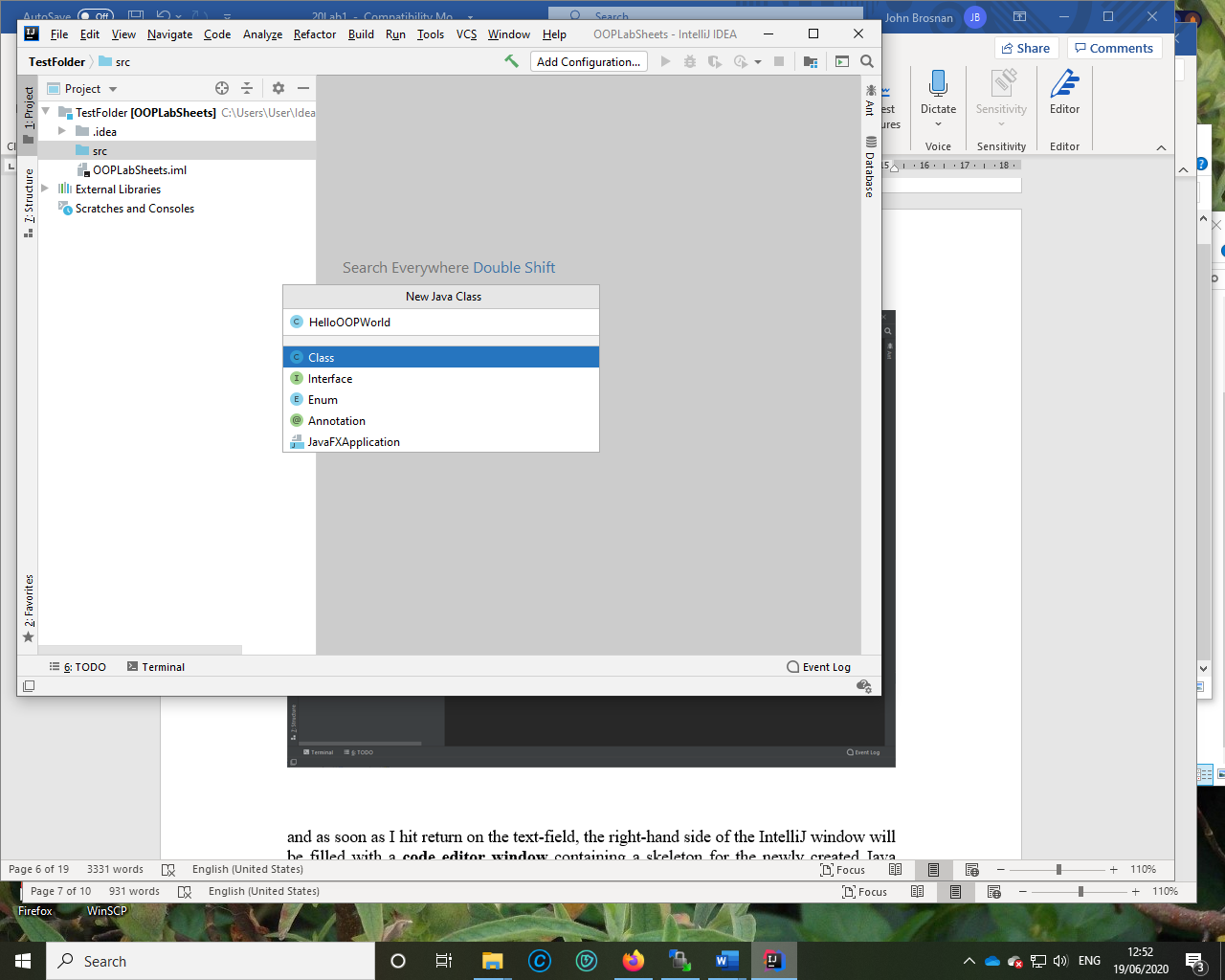


I have entered “**OOPLabSheets**” as the project name and this automatically gets filled in for the last part of the project location (which will default in my case to C:\Users\User\IdeaProjects\OOPLabSheets, since this is where IntelliJ was installed). You could of course save the location of the project to any location though, but I do recommend that you work from the C: drive or memory stick because the **X: drive does have known issues**. You can copy your project to the X: drive as a backup no problem.

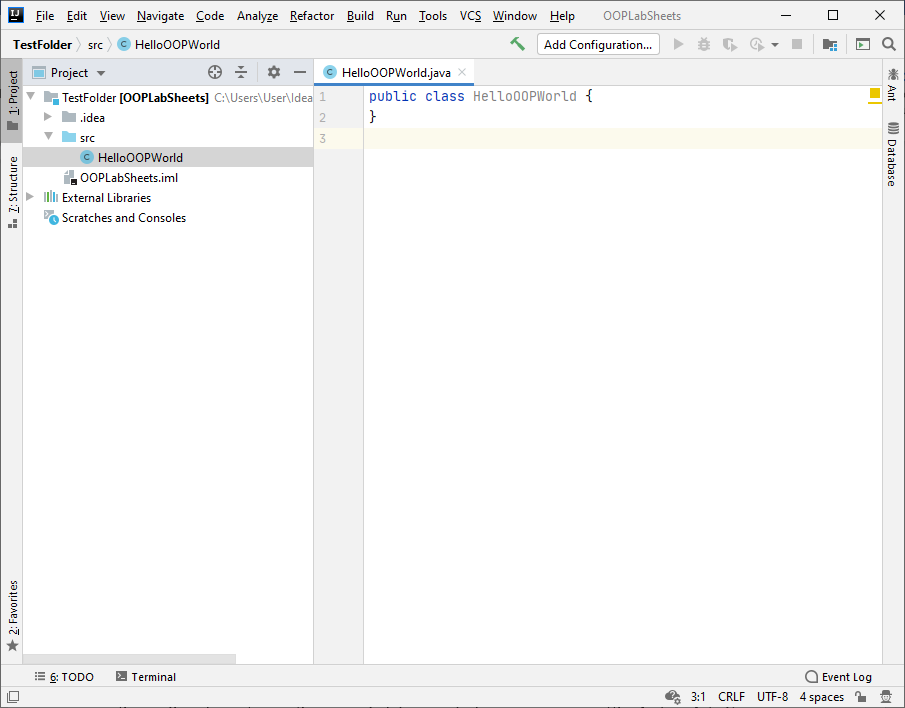
Now click the “**Finish**” button and the Java project called “OOPLabSheets” will now be created and you will see a screen similar to the following:



Now that the project is created, it’s time to add a Java file to it. To do this just **expand** the “OOPLabSheets” project contents in the left-hand side. You will see a **.idea** folder, an **src** folder and a file. **Right click** on the **src** folder and then select **New**🡪**Java Class**. A window will pop up and you will be given the chance to enter a name for your Java class. I have chosen the class name below as **HelloOOPWorld**



and as soon as I hit return on the text-field, the right-hand side of the IntelliJ window will be filled with a **code editor window** containing a skeleton for the newly created Java class. The name of the Java file, **HelloOOPWorld.java**, appears in the tab at the top of the code editor window and you can see in the left-hand side that the newly created Java class now forms a part of the project, within its **src** folder.



You will see a **little green hammer** at the top of the IDE – you can use this to **build** your project (**compile** it). When you do this, IntelliJ will create an **out** folder automatically and this will end up storing the compiled Java code for your program (the **bytecode**). You will see this folder as an orange folder in the left-hand side of the window above the **src** folder.

To run your program, you can use the **Run** menu option or, even quicker, you will see a **little green triangle** near the main() method within the code editor and you can just click on this to run. This assumes your program contains no syntax errors! You cannot run the program above yet though because it has no main() written. To create one just type **psvm** and you can select the correct option to give you your main() in a split second! I think ye are going to like IntelliJ, even if ye don’t end up liking OOP 😊

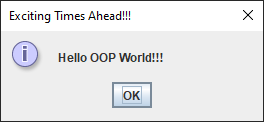
There are heaps of settings and options available for IntelliJ, just **experiment** with them as the weeks go by and see what you can pick up to help yourself in the coding process. One of the big advantages you will notice is that the **Java API is linked to IntelliJ** so you get “**predictive text**” when coding to prompt you. This is especially useful if you are a bit iffy on the name of a method or on how it should be called.

At this point you are ready to code!

**Before you leave the lab, if you have been working from the C: drive throughout the session, do make sure to copy your OOPLabSheets project to the X: drive or memory stick before you leave, otherwise your work will not be available to you.**

**Exercise 1**

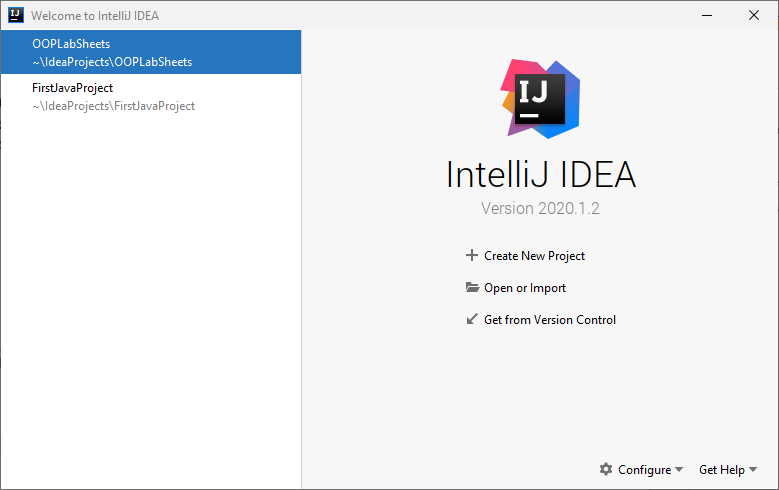
Now that you have your first IntelliJ project created and your first (almost empty) Java file, see if you can add the necessary Java code to the file to produce the following output



In terms of **saving the file**, it is done automatically every time you compile. So, in this case, the file HelloOOPWorld.java is saved to C:\Users\User\IdeaProjects\OOPLabSheets\src on my system. However, as we must create a project in IntelliJ to house our Java files, we will be more interested in the location of our project. In my case here the project **OOPLabSheets** I created originally is on my file system at C:\Users\User\IdeaProjects. So when moving from one machine to another, it is best to **copy the entire project** rather than individual files, since various configuration settings etc. are stored for the project also through the .idea folder and the .iml file attached to the project. When you wish to work on a project you had previously opened, you can just use the **File**🡪**Open** option in IntelliJ, I will get you to try this now as an exercise.

**Exercise 2**

Close the project OOPLabSheets now using the **File**🡪**Close** option and now try to open it again from the start-up window using the **Open or Import** button on the right to **navigate** your way to its location



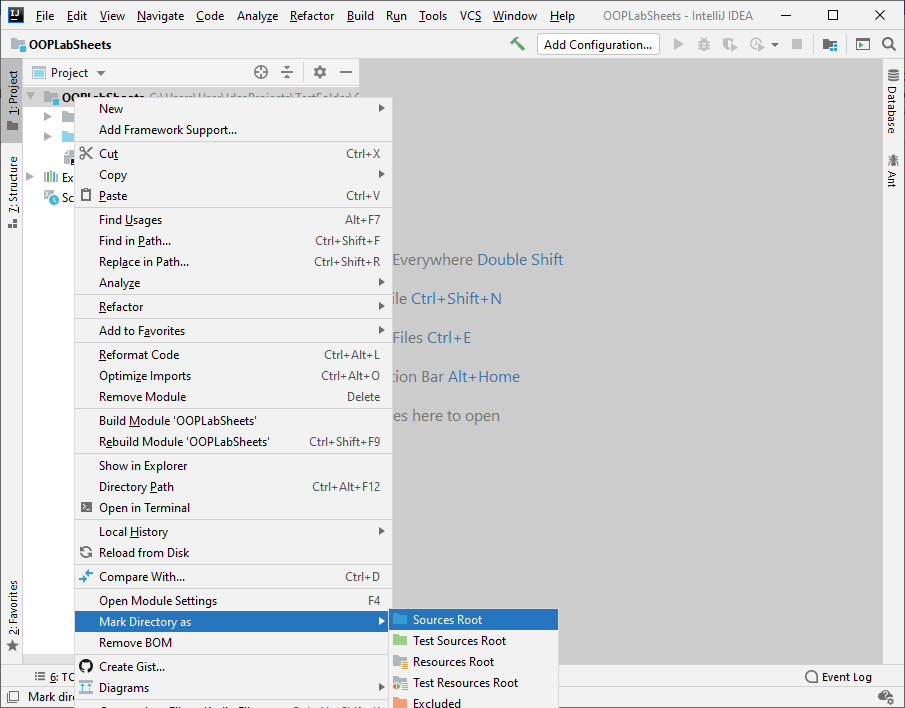
Once opened, the project will appear again in the IntelliJ window and it will display the contents of the Java file in the code editor window as before. Just to make sure the project is still working okay, **build it and run it again**.

**Setting up your Folder Structure**

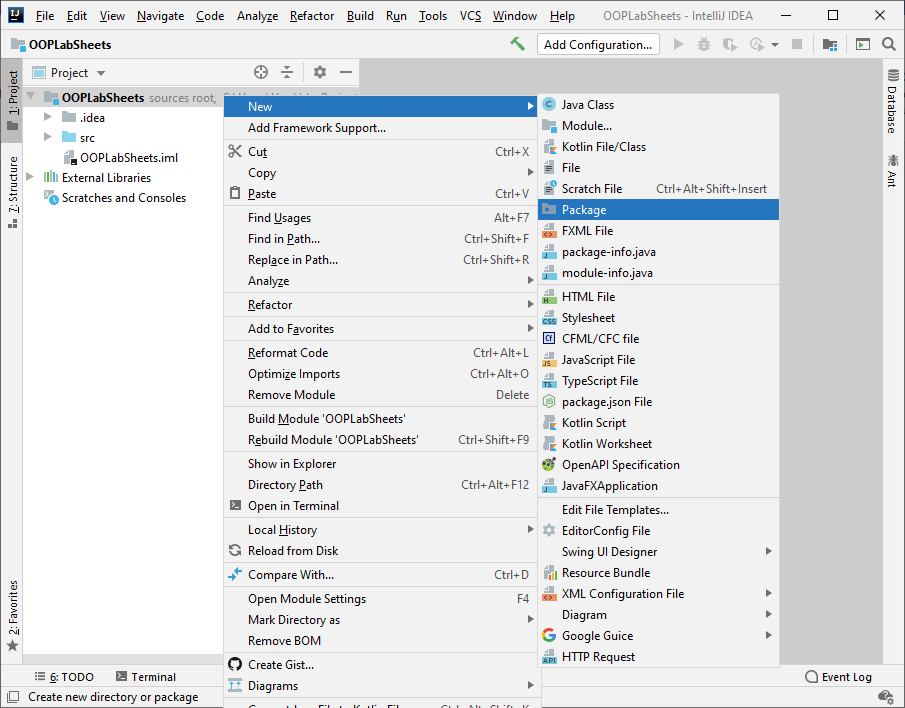
There are many ways to organise your work when using IntelliJ. Some students prefer to create a brand new project for every lab sheet they do, which is perfect, while others like to create a new folder within the overall “lab sheets” project to house the exercises/sample programs from a particular lab sheet. Yet other students just put every single program into the one project, which can work too, but is not so organised 😊

My own preferred approach is to create a new folder for each lab sheet. In IntelliJ this can be done by adding a new **package** to the project. The package will be given an appropriate name, I will call it **labsheet1** here. Of course, you met packages in first year almost right from the start e.g. **java.util**, **javax.swing** etc. are examples of Java API packages. As mentioned in year 1, **a Java package is simply a way to store related classes together** and essentially a **package is just a folder**. We will talk about packages further in this module but for now, we will just create a package called **labsheet1** for this IntelliJ project and our intention will be to store all the related classes that we create and use for this lab sheet together within that package (folder).

First of all, right-click on the project name in the left-hand-side panel and then, in the options that appear select “**Mark Directory As**” and then select “**Sources Root**”. This **allows us to add packages directly** to the project.



Now right-click on the name of the project i.e. **OOPLabSheets** and select **New**🡪**Package** as follows:



You will now be given the opportunity to enter the name of the package, so you can enter **labsheet1**.



As soon as you click OK, an icon for the newly created package appears in the left-side window, listed as part of the project’s contents. The package is currently empty but now we would like to move the file we created earlier, HelloOOPWorld.java, to this package. This can be done simply by dragging it from its current location (the **src** folder) to the newly created package (or you could **cut and paste** the file instead). Press the **Refactor** button on the window that appears. Once you have done this you will see the file HelloOOPWorld.java now appearing within the package labsheet1 as required. Within the Java code you will now see the following additional line at the top

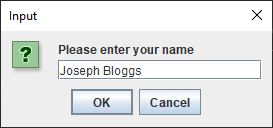
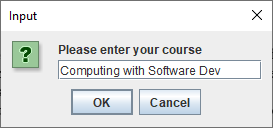
package labsheet1;

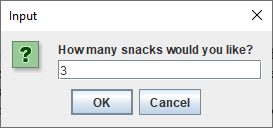
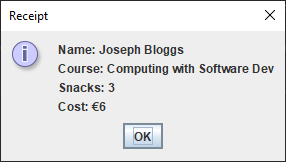
IntelliJ adds this automatically once the file has been moved to the new package. This line of code is a **package statement** and **must always appear before any other line of Java code in a program** (except for comments).

So we have organised our code a little now anyway. Every program you write as part of this lab sheet will now be added to this package. So, for your next (and subsequent) revision exercises, you will just need to right-click on the **labsheet1** package and then select **New**🡪**Java Class** to create a brand new Java file.

**Exercise 3**

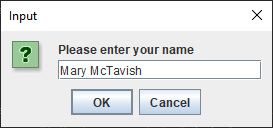
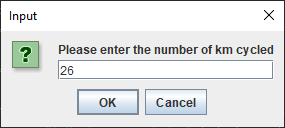
The ITT canteen is offering a special start-of-year mineral-and-muffin snack deal at €2 per snack. Write a program called **SnackDeal.java** which will ask you to enter your name, your course and how many snacks you’d like, and which will then calculate and display in a message dialog your name and course, the number of snacks you asked for and what you will have to pay. Your program output will look as follows:

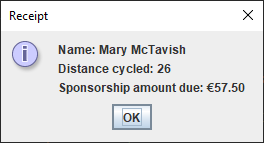
 

**Exercise 4**

A cyclist is sponsored at the rate of €1.75 for each km up to 10 km, and €2.50 for each km in excess of 10km. Use **constants** here to store the two different rates. Write a program called **CyclistSponsor.java** which will ask for the cyclist’s name and the distance cycled (take it that it’s a **whole number** of km here). Your program should calculate the money due, then display a dialog containing the cyclist’s name, the distance cycled, and the money due, to 2 decimal places. Note here that the problem involves **2 different rates** if the number of km exceeds 10 so, for example, if the cyclist is sponsored for 16km then the first 10km are sponsored at €1.75 per km and the remaining 6 km are sponsored at €2.50 per km giving a total amount due of €32.50. The program will run as follows:



**Exercise 5**

On Earth, the acceleration due to gravity, g, is 9.81 m/s/s. It is possible to determine the acceleration due to gravity on a different planet or moon, gp, through the formula

where

* is the mass of the other planet
* is the mass of Earth
* is the radius of Earth
* is the radius of the other planet

Write a Java program called **PlanetGravity.java** that determines the acceleration due to gravity of a planet, based on user-supplied values for the 4 quantities above. This should then be displayed to **2 decimal places**. You should make g a **constant** in your program and set its value to 9.81 m/s/s.

Your program should execute as indicated in the sample run below:

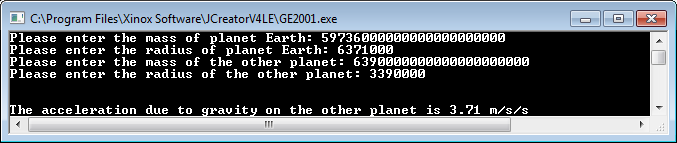
The values I entered are as follows – **ctrl-c/ctrl-v** these into your IntelliJ console when running the program

59736000000000000000000

6371000

6390000000000000000000

3390000



**Exercise 6**

A program called **HeightStats.java** is required that will read in a set of 6 user-supplied adult height values using a **while** loop. Each height value entered is first of all tested to see if it valid. The shortest adult ever recorded was 0.5464m and the tallest adult ever recorded was 2.72m. Therefore, any height value entered must be within these lower and upper limits in order to be valid for this program. Should the height value entered turn out to be invalid, then the user will be given an error message and asked to re-enter the height continuously until a valid one is supplied (use another **while** loop for this validation process).

The program should then process each valid height entered and determine

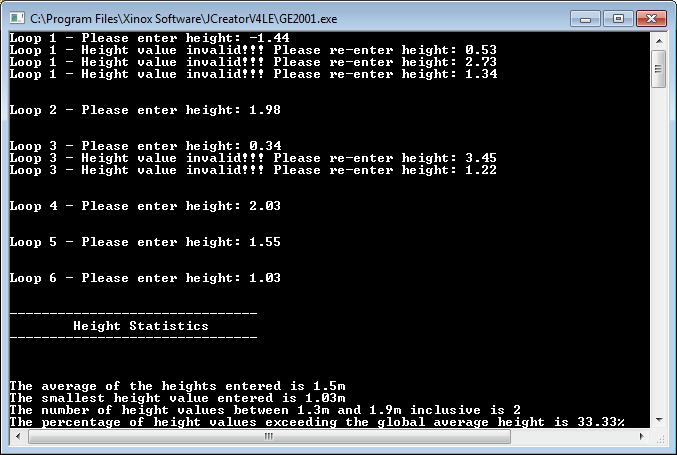
* The average of the heights entered (to 1 decimal place)
* The smallest height value entered
* The number of height values that were between 1.3m and 1.9m **inclusive**
* The percentage of height values that **exceeded** the global average height, which is 1.665m (to 2 decimal places)

You should create a **constant** in your program called GLOBAL\_AVERAGE\_HEIGHT and set it to 1.665.

Using the test values as indicated in the screen shot below, the program should give you **exactly** the following output when it runs, including any banners, blank lines, tabs etc.

**Sample Screen Shot**

**The main loop iterates for the first time and the user enters some invalid height values to begin with, receiving an error message each time through the validation loop. Finally a valid height is entered (1.34) and the program continues on to process this value and then begin the second iteration of the main loop. The second height value entered is valid and so the third loop begins. Some invalid values are entered this time until a valid one is supplied (1.22). Then in the last 3 iterations, the user supplies good height values. Once the main loop finishes the results are displayed.**



**Exercise 7**

A program called **TextAnalyser.java** is required that will read in exactly 3 pieces of text, one by one, using a counter-controlled **do-while** loop. The program will determine and display the following information for each piece of text entered:

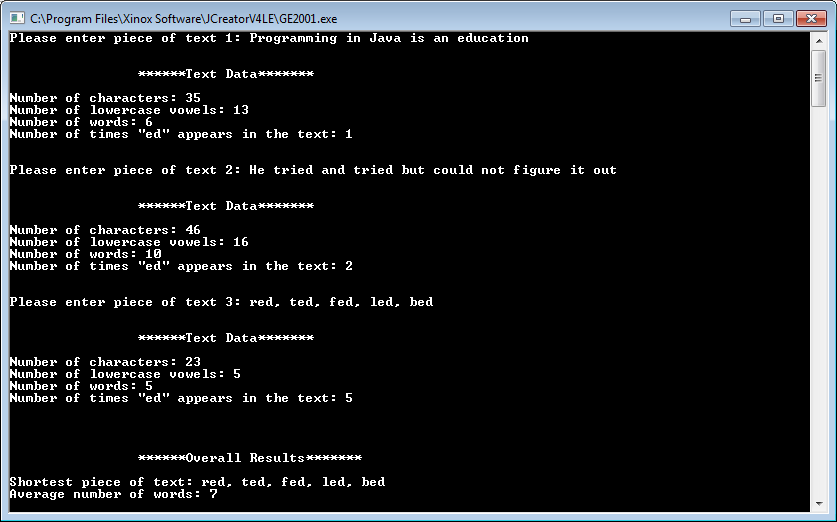
* The number of characters contained in the piece of text
* The number of **lowercase vowels** in the piece of text (use a **while** loop for this part)
* The number of **words** in the piece of text (use the same loop as the last part, we will say here that there is exactly one space between each word so the total number of spaces will be exactly one less than the number of words in the text)
* The number of times the sequence of characters “ed” appears in the piece of text (same loop again for this part, just ensure you don’t go beyond the end of the text here)

When all 3 pieces of text have been entered, the program will then display the shortest piece of text entered and also the average number of words contained in the 3 pieces of text, to the **nearest whole number**.

Using the test values as indicated in the screen shot below, the program should give you **exactly** the following output when it runs, including any banners, blank lines, tabs etc.

**Sample Screen Shot**

**Run 1 – When the user enters each piece of text, some data in relation to that gets displayed. Then, when all 3 pieces of text have been entered, some overall information relating to the 3 is displayed.**



**Exercise 8**

Write a Java program called **NumbersMenu.java** that will use a counter-controlled **do-while** loop to read in **up to** 10 whole numbers, one at a time.

For each number entered, the user will then be presented with a menu of options relating to what they might like to do with the number entered as follows

1. Determine if the number is odd or even
2. Find the factorial of the number
3. Quit the program

When the user has made their choice, a **switch** structure should then be used to decide the course of action the program will take from that point.

Note that the **factorial** of any number n is given by n! where

n! = n\*(n-1)\*(n-2)\*(n-3)\*…..\*3.2.1

so, for example, the factorial of 6 is 6\*5\*4\*3\*2\*1 = 720

Note also that you **cannot find the factorial of a negative number**, so issue an appropriate error message should the user try to determine this.

You should use a **for loop** when determining the factorial of the number in this program.

Your program should run as indicated in the sample screenshots below, which show four iterations of the main loop, before the user then decides to quit the loop early by choosing option 3.

