**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

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

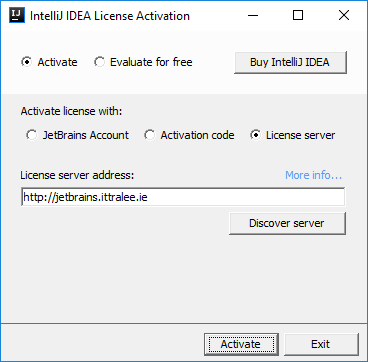
**Practical 2 – Structured Programming Review**

This lab sheet just continues from where we stopped last time, reviewing the various Java syntax and structures and giving you the chance to recall and apply some of the problem-solving routines we covered in first year. Hopefully you will remember a good chunk of it but, if you did struggle with Java last year, pull out all the stops now to try to finally get to grips with it and be sure to **put in some extra hours** beyond class time if you need to. After this lab sheet, we move on to brand new stuff next week!

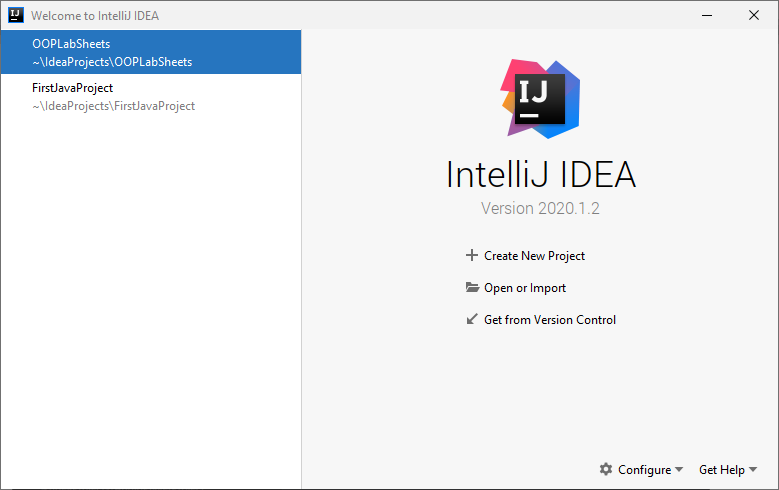
**Getting into IntelliJ**

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**



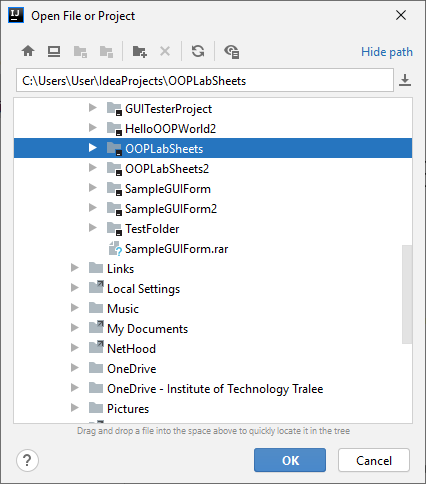
When completed, the IntelliJ **Integrated Development Environment** (**IDE**) should launch for you, after a few seconds. Once this happens, you are ready for coding!



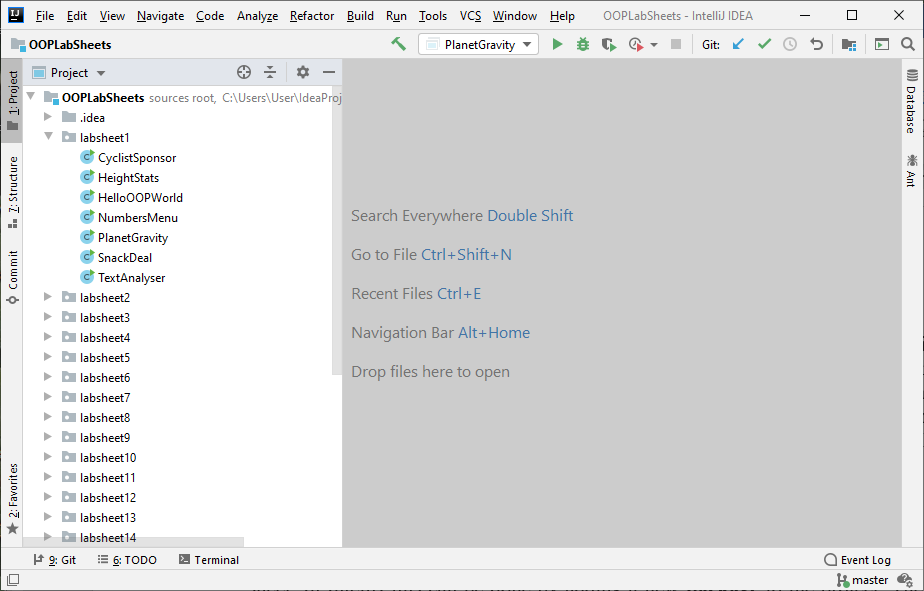
To begin with above, you will see options to “Create New Project”, “Open or Import” etc. Click “**Open or Import**” and try to navigate your way to the location where you have stored the project you created last time called **OOPLabSheets**. It may be on your X: drive or memory stick. If it is on X: drive, then I suggest you copy the project to the desktop and then just navigate to it (as working from X: can be iffy at times). If you are working from memory stick you can work directly from there if you wish and just navigate to the location of the **OOPLabSheets** project on your memory stick.

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

You can see below that mine is located on the C: drive of my machine at C:\Users\User\IdeaProjects. This is the default location for newly created IntelliJ projects.



Once I select the project I want, I just click OK and then the project opens within IntelliJ as follows:



You can see that the project contains a folder called **.idea** and there is also a file called **OOPLabSheets.iml** – these are just for **configuration** purposes and, as programmers, we normally have nothing to do with them. Our main focus would be the contents of the package **labsheet1** (I have clicked on this above) that we created to store all the work we did for the last lab sheet. You can see that within this package are the seven Java files that made up the last lab sheet.

**Setting up your Folder Structure**

As I mentioned last time, my 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 **labsheet2** here. Recall that **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 **labsheet2** 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). Right-click on the name of the project i.e. **OOPLabSheets** and select **New**🡪**Package.**

You will now be given the opportunity to enter the name of the package, so you can enter **labsheet2**. 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 you will be adding a new Java class to it shortly for your first exercise.

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 **labsheet2** package and then select **New**🡪**Java Class** to create a brand new Java file.

At this point you are ready to code!

**Exercise 1**

The details of an arbitrary number of computers are to be processed by a Java program called **ComputerData.java**. The details to be entered are the computers *serial number* (which can contain letters), its *hard disk space* (a whole number of GB), its *processor* type and its *price*. When all the computer details have been entered, the user of the program should then hit return on the keyboard for the serial number (i.e. enter the **empty string**) to indicate the end of input.

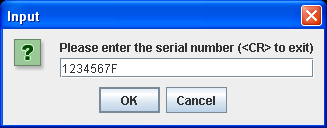
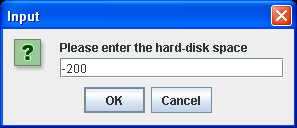
The program should keep track of the total number of computer records entered and use this to display the **average hard disk space** available on the computers to the **nearest whole number**. It should also display the **price range** of the computers entered to **2 decimal places** e.g. €800.00 to €1400.50 along with the **processor type** for the **cheapest** computer.

Your program should ensure that the **hard disk space** of the computer is **validated** perfectly, giving the user as many goes as necessary to enter a valid value and not moving on until such a value has been supplied. You can take it here that only hard disk space values greater than 50 but less than 5000 are valid and that the program should reject any values containing anything other than digits as indicated in the screenshots below.

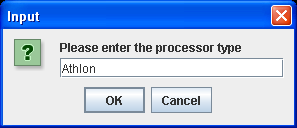
Also, if the user should hit return immediately when the program runs, the program should just terminate at that point with a “No data entered!” message and not display any output results at all.

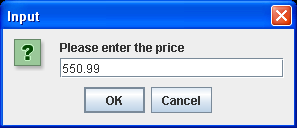
Your program should make use of a **user-defined method** called **averageDiskSpace**() in order to determine and return the average hard disk space value. This method will take **two arguments**, these being the total disk space for all the computers entered and the number of computers processed.

**Sample Screenshots**

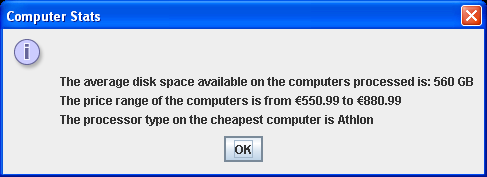
** **

** **

** **

****

**Several rounds of input later …… finally followed by hitting return for the serial number gives**

****

**Exercise 2**

The game of “roll the dice” is to be created in a file called **RollTheDice.java**, which pits you against the computer. In this game a pair of dice are “rolled” for both the computer and you, the player. When a pair of dice are rolled the sum of the values on each face of the dice will give a number between 2 and 12 inclusive. You will be using the **Math** class’ random() method to generate these dice rolls in a random fashion. If the sum of the dice roll for the computer exceeds that for the player, then the computer wins the game, if the sum of the dice roll for the player exceeds that for the computer then the player wins the game, otherwise the game ends in a draw.

Your main() will contain a data-sentinel controlled **while** loop that will allow the user to play as many games of “roll the dice” as they wish. At the end of each game, the user will be asked whether they wish to play another game. If they enter “no” then the loop finishes and the application will give the user a farewell message thanking them for playing the game. If the user enters “yes” then the loop iterates again and another game of “roll the dice” will take place.

After each game is played, a message dialog will appear which tells the user how many games have been played at that point as well as how many games both the computer and the user have won, and how many games have been drawn.

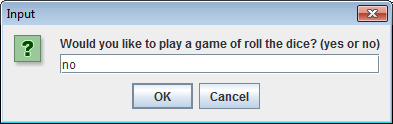
**rollTheDice**() will be a user-defined method that will take **no arguments** at all and will use the random() method twice within it to generate the dice roll for the computer first of all and then the player – these values will be displayed in turn on message dialogs. The dice roll number generated for the computer will then be compared with that for the player and the method will simply return the winner of the game as a **char** as follows: ‘c’ for the computer, ‘p’ for the player or ‘d’ for a draw (when both rolls generate the same number).

Back in the main(), when the rollTheDice() method returns its outcome, there will be code which keeps track of the number of games played, as well as the number of wins recorded for both the computer and player, along with the number of draws.

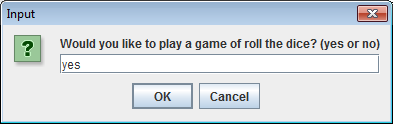
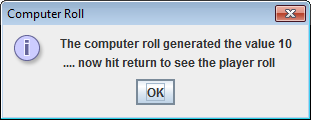
Your program should run as indicated in the following sample screenshots. Of course, as random number generation is used in this program, your output will not match that in my own screenshots below exactly.

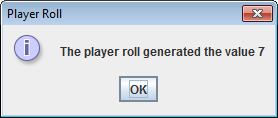
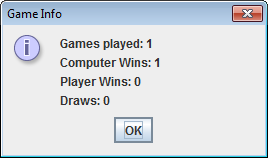
**Sample Screenshots**

**Run 1 – user decides they don’t want to play a game at all here, they just get the farewell message dialog straight away**

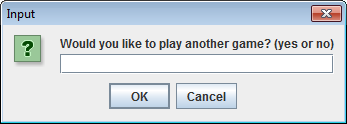
 

**Run 2 – user decides to play a game, a message dialog appears indicating the value of the computer dice roll, then another indicating the value of the player dice roll, then a final one indicating the current game statistics**

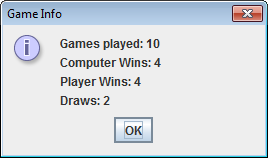
 

**Once the statistics dialog disappears, an input dialog appears again to ask the user if they want to play another game:**



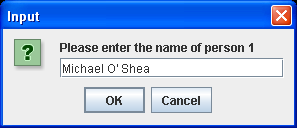
**The process then repeats until the user eventually decides they’ve played enough games by entering “no” on this dialog, at which point the “farewell” dialog appears. The message dialog below shows what the games statistics looked like in my case after playing the game 10 times – yours would be different to this of course, depending on the random numbers generated.**

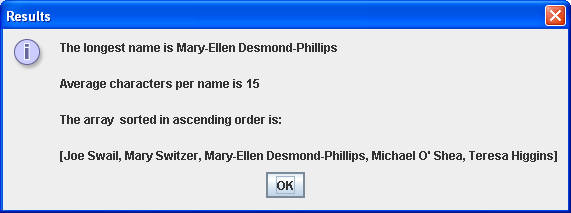


**Exercise 3**

A Java program called **NamesArray.java** is required that will first of all populate a String array called names with some user-supplied values, through a **user-defined method** called populateArray(). You can take it here that the size of the array is 5. The program should then determine the person with the longest name and also determine the average number of characters per name correct to the **nearest whole number**. Finally the program should sort the array into ascending order and then display the names, using the sort() and toString() methods from the Arrays class for these parts. The information then gets displayed on a message dialog as indicated in the sample screenshots below:

**Sample Screenshots**

 **………followed by another 4 rounds of input**



**Exercise 4**

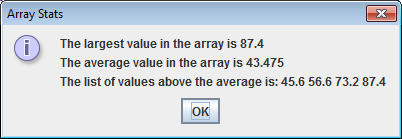
Write a Java program called **NumbersArray.java** that will first of all initialize an array of double with the following eight values:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 22.3 | 45.6 | 27.4 | 56.6 | 73.2 | 11.5 | 87.4 | 23.8 |

The main() will then call on three **user-defined methods** to carry out three separate tasks as follows:

* A method called **largest**() which will take the array as an argument and return the largest value found in the array
* A method called **average**() which will take the array as an argument and return their average
* A method called **aboveAverage**() which will take both the array and the average determined earlier as arguments and return a String which contains a list of all those values in the array that exceed the average

The values returned by these three methods will then be displayed in the same message dialog, as indicated in the screenshot below - note that the average gets displayed to **3 decimal places**.



**Exercise 5**

A Java program called **RandomArray.java** must be written that begins by creating an array of int of size 10. main() then calls a separate user-defined method named **populateArray**() whose job is to fill each slot of the array with some random integer number between 1 and 1000 inclusive. This method contains a for loop that will iterate 10 times. The random numbers will be generated with the help of the **random**() method from the Math class.

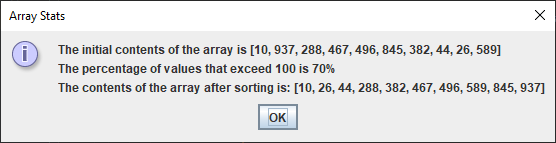
When the array has been populated with values, the main() should then:

* Display the initial contents of the array
* Sort the array into ascending order
* Determine the percentage of values in the array that exceed 100
* Display this percentage and the contents of the array after sorting

You are welcome to use the appropriate methods from the **Arrays** class when coding this program

**Sample Screenshot**

**Note that there is no user input in this case as the array is populated automatically through the generation of random values**



**Exercise 6**

Write a Java program called **WeightConverter.java** that contains a JFrame window. This window should have a 2 labels and a text-field (of size 5). It should have dimensions of 300 x 100 pixels. It should use a flow-layout style and the application should terminate when the close button is hit on its title bar. The first label should simply prompt the user for their weight in pounds. This should be input by the user into the text-field. When the user hits return on the text-field, having input a value, the equivalent weight in kilos should be displayed on the second label, corrected to **2 decimal places**. Hitting return on a text-field constitutes an ActionEvent, just like pressing a button, so treat it in exactly the same manner as this code-wise from the point of view of event-handling.

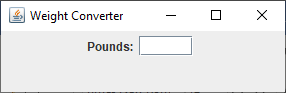
Note that there should be a little **validation** in the application. For example, if the user hits return on the text-field but no value has been entered into the text-field, a suitable error message in a dialog should appear to the user. No other validation whatsoever is necessary, however.

**N.B.** **1 pound = 0.454 kg**

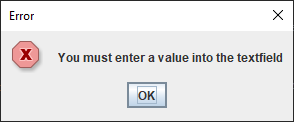
Some sample runs of the program are as illustrated below.

**Sample Screenshots**

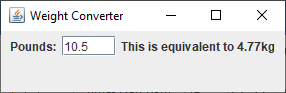
**After launching the application it appears as follows:**



**If the user hits return at this point the following error dialog appears:**



**If the user now types in a quantity in pounds and then hits return the following happens:**



**Exercise 7**

Write a Java GUI application called **NumbersGUI.java** containing a JFrame window that has 3 labels and a text-field (of size 20). The JFrame window should be 500x150 pixels in size and make use of a flow-layout style. The application should simply terminate on closing the window.

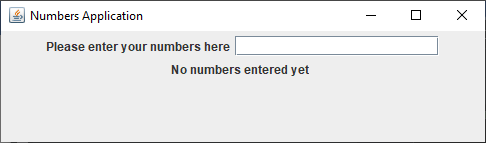
One of the labels will prompt the user to enter numbers. When the user enters a value and hits return on the text-field an ActionEvent will be triggered. Every time this event occurs, the program should display the current largest and smallest numbers entered by the user, on a second and third label respectively, and clear the contents of the text-field. When the GUI launches initially, the second label should just display the text “No numbers entered yet”, with the third label just remaining blank, as indicated in the first screenshot below.

Note that hitting return on a text-field constitutes an ActionEvent, just like pressing a button, so treat it in exactly the same manner as this code-wise from the point of view of event-handling.

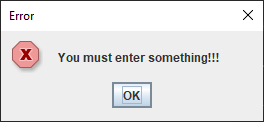
If the user enters nothing at all and proceeds to hit return on the text-field, a message dialog should appear issuing the message indicated in the second screenshot below.

Some sample runs of the program are as illustrated below.

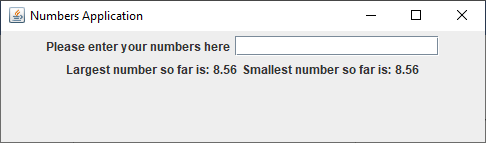
**When the application first launches:**



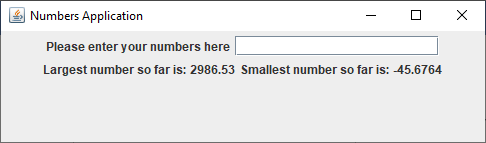
**If the user enters the empty string:**



**If the user enters the value 8.56 to begin with – note that the text-field gets cleared of the value just inputted and the 2nd and 3rd labels display information**



**….. many input values later**

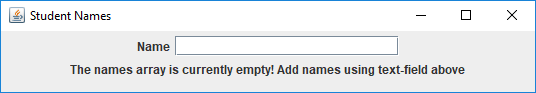


**Exercise 8**

A Java program called **NamesGUI.java** is required that will first of all create an array called names which is capable of storing five student’s names. This array will then be processed via a GUI interface.

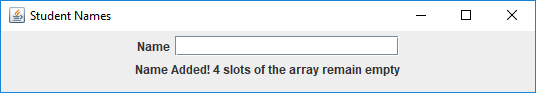
The main GUI interface itself consists of a JFrame window. It should have dimensions of 500 x 100 pixels. It should use a flow-layout style and the application should terminate when the close button is hit on its title bar.

To begin with, when the GUI launches, it will contain 2 labels and a text-field (of size 20) and you should see the following interface

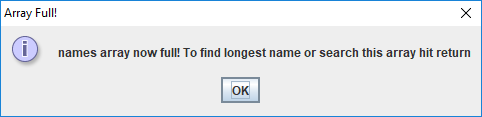


At this point the names array is empty and the user is informed of this on the second label. They are also prompted to populate the array by using the text-field on the same label.

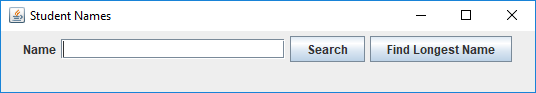
Then, when the user enters a name on the text-field and hits return on it, the name entered gets added to the first slot of the array, the text-field gets cleared and the user gets confirmation that the name has been added to the array and how many slots of the array remain empty (use a **global counter variable** for this, which increments each time an event occurs on the text-field), via the second label as indicated below



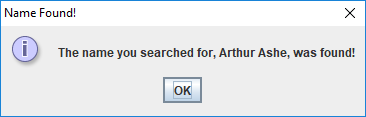
Finally, when the user has added five names to the array, a message dialog pops up to indicate that the array is now full, as follows (your code should ensure that, if any further events occur on the text-field, they will not be processed – you can use the same **global counter variable** mentioned earlier to achieve this)



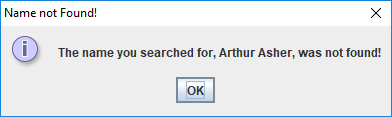
When the user hits return on this dialog, the GUI interface changes. The second label is made invisible (call **setVisible(false)** on the label reference to achieve this) and two buttons are added to the interface. The first of these is the “Search” button, which will be used to perform a **binary search** on the array for a particular name (entered by the user on the text-field) and the second will be a “Find Longest Name” button, which will be used to determine the longest name in the array. So the GUI now looks as follows:



When the user presses the “Search” button, the name currently entered into the text-field will be retrieved and then checked to see whether or not it is in the array of names, via a **binary search** **algorithm**. Recall that you must **sort** the array before you can perform the binary search algorithm – for this question, feel free to use the **sort**() method from the **Arrays** class to accomplish this (if you are feeling adventurous, you can try to write the selection-sort code instead from first principles of course 😊). In this case, after the binary search algorithm completes, a message dialog will display to indicate whether or not the name entered was found. For instance, in the following case, the name was found in the array



While in the next case, the name was not found



You should try to code your search algorithm within a separate user-defined method called **binarySearch**(). Note that this method should take the **names** **array** and the **search name value** as arguments and return a boolean true or false result to indicate if the search value was found or not.

When the user presses the “Find Longest Name” button, code executes which loops through the names array to determine the longest name entered, and then displays this on a message dialog

