Worksheet 5: Modularity and More Arrays

Updated: 26th March, 2021

The objectives of this practical are to:

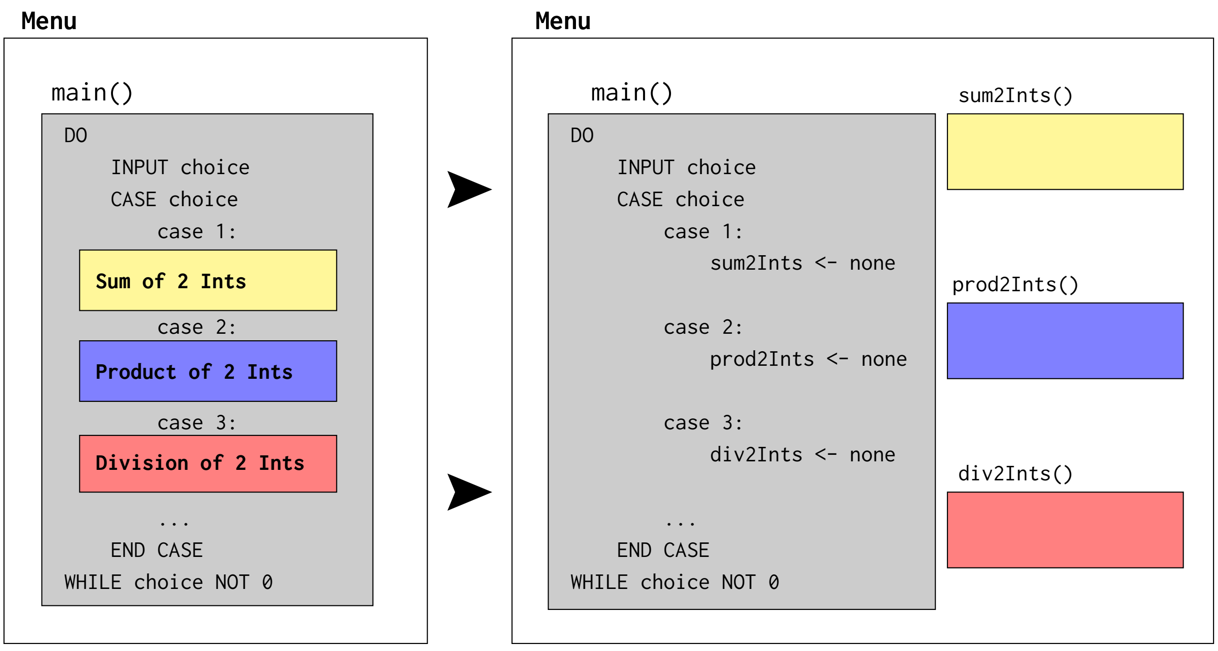
- practice modularisation;
- revise previous concepts;
- practice test-style questions.

1. Introduction to submodules

By now, our **Menu** has gotten quite large and is contained all within a single submodule – `main()` Yes, `main()` is a submodule. Depending on how you approached the problem, the structure of your code should be similar to that shown on the left below.

Now, it is time to modify and adapt your **Menu**, so that it becomes more modular and makes effective use of submodules.

Note: Your menu might be especially large, depending on the instructions you've received from your tutor. For this week, you are free to remove menu options that are not **1**, **2**, **3** or **0**. These remaining options must still work, however.



Create a copy of the **Menu** pseudocode, test plan and Java code from **P04** into our **P05** folder for this week. We will be building upon this for today's practical.

Modify your pseudocode by separating the **main** in to several submodules, one for each "task". The structure of your program should now resemble that shown in the diagram above. Your **main** will still contain the menu input and selection, but now each menu option should just call the relevant submodule. Remember to follow the naming conventions and all good practices outlined in the lectures.

Note: Readability of your pseudocode is just as important as that of your Java code. For now, the below is a good set-up to use for your submodules; you should maintain indentation in pseudocode too.

```
SUBMODULE: moduleName
IMPORT: variableName1 (DataType), variableName2  (DataType), ... etc. or none
EXPORT: singleVariableOnly (DataType) OR none
ASSERTION: "To retrieve some sort of result, or to do some task."
ALGORITHM:
    ...
    ...
```

As per usual in this unit, you are free to use your own pseudocode style, as long as it complies with the 'CLUCC' standard. The above is a suggestion that you **may** wish to use.

2. Further modularising code

Separating distinct tasks into their own submodules is one purpose of modularisation. However, if you examine these new submodules, they still perform multiple tasks: input, calculation (or processing) and output.

We will deal with the easiest of these first – the calculation. Create a submodule for each of the calculations within your **Menu**. Each submodule should **import** the necessary values, perform the calculation, and **export** the result. These submodules seem fairly basic, due to the scale of this program, but you will need to learn how to make your code modular for future algorithms.

Note: You may ignore the **INPUT** for now, as we will cover how to make that modular in the next part of this worksheet.

It may be tempting to leave the calculation in the same submodule as other steps. It is **very** important that the submodule imports the values, performs the calculation, and exports the result!

In upcoming practicals, we will be adding an option for the data to come from the user, or from a file, as well as for the result to be given back to the user or written to a file. If you do not have the calculation as its own submodule, you will have problems implementing this functionality and have to rewrite your code!

Your **Menu** algorithm should now be broken up into many submodules each doing one specific calculation task. You will, however, have some 'leftovers' for when the user inputs (enters) data or data is output (printed) to the screen. Ensure that if you have any repetition of code in your submodules (excluding single line print statements) that they are put into their own submodules now. Hence, input and output statements should now be placed into theirown submodules.

It is a good idea to check with your tutor, with regards to the amount and complexity of each submodule. Those of you taking "Introduction to Software Engineering (ISAD1000)" should reflect on the coupling and cohesion of each submodule.

The name of our class (**Menu**), although correct, is a bit lacking when it comes to being descriptive.

Our task now is to change the name of our **Menu** to **PDIPortfolio** to showcase our work that we have created.

3. More modular menu modifications

It is now go back to your newly created submodules, have a look at their algorithms. Are there any consecutive lines that are very similar? What about all the times we require **INPUT** from the user?

Again, those lines should look fairly repetitive. You may say that "I output different prompts!" – that is fine. If you think about it, your output string is just an argument you are passing to a method (e.g. `System.out.println()`).

> **Note:** In pseudocode, we do not usually worry too much about data types, however, we just ran into one of the few situations where it could matter. Each of our tasks requires **Integer** values from the users, however quite often we will need real numbers or character values to be entered by the user.
>
> In time, this means you will need **three** submodules for input; one for inputting an **Integer**, one for inputting a **Real** and one for inputting a **Character**. Of course, only those required would need to be included.

The input submodules are another example of code re-usability; the input of every **Real** number is now dealt with in a single method. The input submodules should contain the loops necessary to ensure the value is **valid**, meaning that the **EXPORT** of the submodule is guaranteed to be valid.

Similar to last week, below is a (slightly modified) example of an **INPUT** submodule:

```
SUBMODULE: inputX
IMPORT: prompt (String), min (X), max (X)
EXPORT: value (X)
ALGORITHM:
    value = min - 1
    error = "Error: Value must be between " + min + "and " + max

    DO:
        OUTPUT prompt
        INPUT value
        IF value NOT BETWEEN min AND max:
            OUTPUT error
        END IF
    WHILE value NOT BETWEEN min AND max
```

We will build on these **INPUT** submodules next week when we cover **Exceptions**. This will allow us to stop the user "crashing" our program when they enter the wrong datatype to our **Scanner**.

There is no need to modify your pseudocode at this time. We will do this next week, at which time we will also convert our pseudocode to Java code and test it.

4. An example of scope

Below is an example of a Java file with submodules in it, showing details of where the scope of variables exists.

```java
public class ScopeImage
{
    public static void main(String[] args)
    {
        int var1;
        if(booleanExpression)  // Assume this is declared
        {
            // var1 is accessible here.
            int myVar;

        } // myVar ceases to exist here.

        // myVar does not exist here.
        // var1 is still accessible here.

    } // var1 ceases to exist here.

    public static void submodule1()
    {
        // var1 cannot be accessed here .
    }

    public static void submodule2()
    {
        // var1 from main cannot be accessed here, either.

        /* This is a different var1 to the one declared in main
           i.e. it is stored at a different location in memory. */
        int var1;
    }
}
```

This is what the scoping looks like for a class with submodules. Notice how local variables declared in the different submodule's scope cannot be seen/accessed by other methods. Each "scope" is defined in a literal sense, by the two curly braces surrounding it. If you need something to be accessed in a different scope, either pass it around or declare it in a higher scope.

Note: Be careful about declaring variables in too high of a scope; once they leave a method, they are considered "Global Variable"s - this leads to zero marks being applied to the entire assessment. Do remember that **main()** is also a method. When you get to creating objects, this becomes slightly less restrictive.

5. Pond storage calculator

Given the following description, you are required to design an algorithm using psuedocode. Create a subdirectory in this week's **P05** directory and name it **PondCalculator**.

Note: While reading the question, think about what submodules you can reuse! This is the benefit of submodules – they can save you time!

Three acquaintances, Joey, Cory, and Rachel, have decided to start keeping aquatic animals in their ponds. They require help figuring out the volume of water they need in order to completely fill their ponds. For each person, you are required to, based on the **depth**, **length** and, **width** of their ponds, calculate the **volume** of water they require. The **depth**, **length** and **width** of the ponds should be input by the user as real values in metres.

Each person has two options for what animal they can keep. Your program should prompt the user to select one of the two options, and loop to ensure a valid option is selected. Then, based on the volume of water and the table below, find out how many animals Joey, Cory, or Rachel could store in their ponds. You do not have to take into account the displacement of the animals in the pond. Output the number of animals, rounded to the lowest whole number.

Your algorithm should contain a looping menu that allows the user to pick whose pond they wish to calculate each time it is repeat, and then which animal that person should keep (you can only keep one species in the pond at once).

Person	Animal	Animals Per m^3
Joey	Sting Rays	0.5
	Arowana	0.4
Cory	Koi	0.6
	Puffer Fish	0.8
Rachel	Turtles	1.2
	Frogs	4.5

The output should look similar to the following:

```
Joey can store 200 sting rays in his 400m^3 pond.
```

Alternative, it could look like:

```
Cory can store 687 puffer fish in his 550m^3 pond.
```

Finally, it could also look like:

```
Rachel can store 182 turtles in her 152m^3 pond.
```

Once you have completed the pseudocode, implement your design in Java. As always, you are required to test your code.

Note: Ensure you are filling out your testing plan. It is **OK** if your code for this question fails some tests, you are still learning, the important thing is to **always** test what you have done so you can understand what has/hasn't worked.

6. Two-dimensional arrays

Following on from one-dimensional arrays, two-dimensional arrays can be used to store groups of the same type of data in a 'tabular' or 'matrix'-style format. This has many uses, such as matrix mathematics (used for computer vision, amongst other things) and panel observation data (such as used in data analysis and analytics of sensor data).

The definition and use of two-dimensional arrays are very similar to those of one-dimensional arrays; instead of one pair of square brackets after the data type, two are used. An easy way to remember which one is which is 'DNA' – down and across, as seen in this example table below:

x	Col 1	Col 2	Col 3
Row 1	`x[0][0]`	`x[0][1]`	`x[0][2]`
Row 2	`x[1][0]`	`x[1][1]`	`x[1][2]`
Row 3	`x[2][0]`	`x[2][1]`	`x[2][2]`

In this exercise, you are to design an algorithm (using pseudocode) that allows the user to enter in two dimensions to generate a two-dimensional array – the number of rows and number of columns. Once the user has entered in the dimensions, a two-dimensional array of the specified size should be generated.

Each value of the two-dimensional array should be equal to the product of its element number; i.e. in the example above, for element `x[1][2]`, its value should be $1 \times 2 = 2$, whereas for `x[0][2]`, its value should be $0 \times 2 = 0$.

Once the values have been calculated, they should be output to the user. To do so, use the `\t` character to generate a 'tab' in your output and the `\n` character to generate a new line for alignment such that the output is readable. For example, the following output would be expected for a three-by-three two-dimensional array:

```
0   0   0
0   1   2
0   2   4
```

Once you are comfortable with your algorithm, implement it in Java and write a test plan to validate it. Execute the test plan by running your program, correcting any errors where discovered.

7. Practice question: Euler's constant

Create a subdirectory in this week's **P05** directory and name it **Euler**.

Euler's constant can be calculated as the sum of the infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

where $0! = 1$.

Design, using pseudocode, an algorithm which will:

- request a valid input of a number of terms to approximate; e.g. your algorithm should repeat the input until the number of terms input is between 6 and `100` (including 6 and `100`);
- for each term, calculate the value of $\frac{1}{x!}$, storing it in an array;
- after all of the individual terms have been calculated, calculate the final value of e, storing it in the last element of the array;
- upon completion of all the calculations, the algorithm should output each value in the array to the user.

Implement this algorithm into a complete Java program. Create and execute a test plan to verify that there are no errors with the algorithm, updating your pseudocode and Java code where errors are found.

8. Practice question: Permutations

Create a subdirectory in this week's **P05** directory and name it **Permutations**.

Write a pseudocode algorithm that determines the number of permutations (**without repetition**) when selecting a set number of elements from a set. This can be calculated as follows:

$$^{n}P_{r} = \frac{n!}{(n-r)!}$$

where n is the number of elements in the set and r is the number of elements you wish to select from it.

Your algorithm should perform the following steps:

- Prompt the user for the number of elements in the set (n); your algorithm should loop until this number is between 5 and 50 (inclusive);
- Prompt the user for the maximum number of elements that should be selected (r_{max}), an integer value between 2 and n (inclusive);
- Create an array of size $r_{max} - 1$;
- Calculate the number of permutations, for each r value in the range 2 to r_{max} (inclusive). The result of each calculation should be stored in the array;
- Once all r values have been generated output the array to the user.

For example, the following outputs can be expected for the following inputs below:

Input n	Input r_{max}	Output
5	4	`[20, 60, 120]`
6	3	`[30, 120]`
8	8	`[56, 336, 1680, 6720, 20160, 40320, 40320]`

Implement this algorithm into a complete Java program. Generate and execute a test plan to verify that there are no errors within it, updating your pseudocode and Java if required.

9. Possible assignment task: Fibonacci series

Create a subdirectory in this week's **P05** directory and name it **Fibonacci**.

This weeks' assignment task is to create a sequence generator. The program should be able to generate a sequence similar to the Fibonacci sequence, but have the ability to specify the first two values and to decide if addition or subtraction should be used to generate the sequence. Your algorithm will need to perform the following steps:

(a) Prompt the user for the mode of generation (subtraction or addition).

(b) Prompt the user for the number of elements that should be generated, an integer between 5 and 50 (inclusive).

(c) Prompt the user for the first two digits of the sequence, integer between -100 and 100 (inclusive).

(d) Create an array of the appropriate size and fill it according to the above information. The first and second elements of the array should be the two digits entered in step 2. Then either add or subtract the two together to create the third element, apply the same calculation to the second and third elements to generate the fourth element.

(e) Provide a looping menu to the user where they can view a specific element of the array, or have the entire array displayed to them.

For example, if the user selects subtraction, 10 elements, and starting digits 2 and 3, then the pattern generated would be `2,3,-1,-2,-5,9,-14,23,-37,60`. If the user then specifies the fourth element should be printed, then -2 should be printed.

Note: While arrays in most languages are 0 indexed, asking an end user with no programming knowledge to understand and use a 0 based index is extremely detrimental to usability. Hence the user wanting to see element 1, is actually the first element in the array, that is index 0.

You must make good use of submodules in your algorithm, make sure you pay attention to code reuse and don't forget to utilise constants.

Once your pseudocode design is completed, you can implement your design with Java and complete and execute a test plan.

End of Worksheet