010 EXERCISES WORKBOOK

# ARITHMETIC MEAN

*My response to question 2 posed in the task brief is implemented in my solution, whereas the response to question 1 can be found in the block comment at the top of my solution.*

To begin, I declared two variables:

1. ‘result’ as the float value 0 (‘0f’)
   * to handle the fact that the mean value returned would most likely be in decimal form,
2. ‘user\_choice’ as an integer
   * so that my program could handle the user’s decision to either make a calculation or exit the application with greater ease.

I decided to contain the bulk of my code within a DO-WHILE loop so that the code in that block would continue to execute until the user chose to exit the application. I thought that this would be more appropriate than using a WHILE loop as I knew that the code block would need to execute at least once, and WHILE loops tend to only be used when the number of times the loop executes is unknown. I also implemented a TRY-CATCH statement to handle any format exceptions that might be caused by the user’s input, such as if they inputted a letter or symbol instead of a number. This was put in place as an attempt at stopping my application from crashing should one of these exceptions have been thrown. One limitation of using a CATCH (FormatException) statement in my program was that even though it handled exceptions thrown by a non-numeric input from the user, it would not throw an exception if the user entered number outside the menu range (i.e. 3, or 90). As a result of this, I added an IF/ELSEIF/ELSE statement to the program which meant that if the user didn’t input either the number 1 or 2, an error message was displayed and the user was prompted to input a valid menu option.

If the user chose to make a calculation, they were asked how many items were in their data set and the program then initialised an empty array of floats called ‘numbers’ of that size. This was to allow the user to enter all the items in their dataset (whether that was more or less than the five items outlined in the brief), and to also allow them to input decimal values.

I then used a FOR loop and a loop counter variable (‘c’) to iterate over each item in the generated array, prompting the user to input a value each time. The value stored at each index was then added to the ‘result’ variable. I chose to use FOR instead of FOR-EACH as I was only iterating over each array item once. As well as this, if the user had inputted a larger data set, the FOR-EACH loop would have been less efficient as it uses more stack space and more local variables.

If the user chose to exit the application, the user was first informed that the application was exiting and then the exit statement was executed. I used the Thread.Sleep() class to create a pause effect rather than the Console.ReadKey() function to allow the user to read the message before the console closed without the user having to input a key. I then used the Environment.Exit(0) method to terminate all running threads and the process itself to ensure a clean exit from the application.

# RECTANGLES

*I have not answered question one due to the discussion the course had during webinar three.*

In line with the task brief, I created two separates functions called RectangleArea(): one which took integer parameters for length and width, and one that took floats. In the main method, I asked the user for the dimensions of their rectangle and stored their inputs in two separate string variables. I stored them as string variables so that I could use the ‘.Contains()’ method to check whether the string entered by the user contained a period ‘.’. If the string did contain a period, this meant that the number entered by the user was a decimal and therefore the program should convert those strings into float values so that the float RectangleArea() function could be called. If the string didn’t contain a period, the program should then convert the string inputs into integer values to be used in the integer RectangleArea() function as the values entered were not decimal types.

I decided to use float.Parse and Int32.Parse to convert the strings, as by just using the Convert function – an extra check would have been carried out to check whether the inputted values were null or not, which would take slightly to execute.

I used overloaded methods to make it easier for the developer to read my code, as both functions make the same calculations using variables with the same names – with the only difference being the data types of the parameters they took. Although I used overloaded methods, these weren’t explicitly from the systems library. This is because even when the systems library is not declared in the program, the methods can still execute.

# TEMPERATURE CONVERSION

When creating this solution, I decided to create a function called ConvertAgain() to ask the user whether they wished to make subsequent conversions and handle their response. I realised that this code would have been repeated unnecessarily throughout my application, and therefore decided it would be sensible to encapsulate this code in a function which I could then call upon when needed - thus improving the efficiency of my application.

After initially asking the user to input a combination of letters in order to select the type of conversion they required, I decided it was more effective for the user to choose from a numbered menu instead. By implementing a numbered menu, the user was relieved of the burden of inputting the full measurement unit each time - which proved advantageous for error handling. By containing my code in a TRY-CATCH (FormatException) statement, it meant that if the user’s input was non-numeric an exception would be thrown and an error message would be displayed to the user asking them to input a valid menu option. This helped improve efficiency as the user only had to input a single digit integer instead of a string of characters, thus significantly reducing the margin of error. As the FormatException didn’t catch numeric user input errors, e.g. if they entered 10, I then added in a default case to throw an error message to the user telling them that they’d inputted an invalid menu option.

Once the user had inputted the number corresponding to the conversion they wanted, the program used a SWITCH statement to branch respectively. I decided to use a SWITCH statement as generally speaking, they tend to execute more efficiently than multiple nested IF/ELSEIF statements. This particularly appealed to me as the ConvertAgain() function contained nested loops. The user was then prompted to enter the number they wished to convert (which was stored in the ‘temp’ variable), and the program calculated the converted value. I declared ‘temp’ as a float to allow for a greater degree of accuracy in my converted value.

In order to allow the user to make subsequent conversions, I contained my code inside a DO-WHILE loop which executed as long as the user didn’t choose to exit the program. If the user chose not to make a subsequent conversion, the program would break from this loop and exit using the Environment.Exit(0) method after displaying an exit message to the user. This was to ensure that all running threads and processes were terminated so that the application could exit cleanly.

I also decided to provide style formatting in the console. If the user decided to make another conversion the console would clear automatically, thus improving readability to the user as each conversion would be isolated.

# DRINKS MACHINE

After reading the task brief, I decided to use a dictionary to store the remaining quantity of each ingredient instead of an array as I could then associate each quantity with the name of its respective ingredient. I then initialised an integer array of six elements to count the quantity of each drink bought, so that in the UI the user could see how many of that particular drink had been purchased.

I contained my code within a DO-WHILE statement rather than a WHILE statement, as the task brief stated that the program should continue running and offering the menu until the user decided to quit the application (or an ingredient ran out of stock). I then placed a TRY-CATCH statement around my SWITCH statement in order to ensure that the application could handle user errors, as well as reducing the likelihood of the application crashing and unhandled exceptions being triggered. I chose to use a SWITCH statement as, when used for multi-way branching, they are usually more efficient than a set of nested IF/ELSE statements. When the compiler compiles a SWITCH statement, it creates a branch table based upon the case constants outlined which it then uses to decide an appropriate execution path for each particular case. As a result of this, SWITCH statements tend to execute more quickly than IF/ELSE statements as all the cases are given equal access time rather than having to wait for the preceding statement to be evaluated.

Upon my first attempt at coding this application, I realised that within each case of my SWITCH statement a lot of the code was repeated. In order to make my application more efficient, I decided to create individual functions for these pieces of code so that instead of re-writing it multiple times, I could call the functions instead. For example, the code used to tell the user that the ingredients needed to make their drink were out of stock featured in six of the seven cases in my SWITCH statement initially – whereas now the function containing this code (OutOfStock()) is simply referenced six times. I abstracted my functions and methods into a class system to help improve readability and assist in debugging.