Worksheet 4: Looping, Arrays & Exceptions

The objectives of this practical are to:

- practice using loops with a boolean expression (**WHILE**, **DO-WHILE**);
- practice using loops with a fixed number of iterations (**FOR**);
- learn how to use one-dimensional arrays;
- learn how to catch basic exceptions.

1. Adding a loop to your menu

Last week, you created a menu to allow the user to select different functions within a program that they would like to execute from a list of options. Your task for this week is to loop this menu, such that the program only exits (or stops running) when the user chooses to exit the program. As such, the user can choose to run more than one part of the program at a time.

> **Note:** Remember the rules of using **WHILE** and **DO-WHILE** loops from the lecture slides. You may be penalised if you use these structures improperly (such as including **break** statements).

Make a copy of your **Menu.txt** and **Menu.java** files from your **P03** directory in to your **P04** directory.

Your menu should include this option from last week:

```
> 0. Exit
```

Currently, this should only output a goodbye message when selected by the user. Now, you will modify its logic so it will be used for another task. When the user selects this option, the program will stop execution and exit. That is currently the behaviour your program has – but now, in other cases, that will change!

Using what you have learnt in the lecture about loops, adjust your program need to loop your menu, so that it loops **one or more times**. The menu must keep printing to the screen (and thus the menu should loop) until and only until the user selects **0**. If another option is selected, the functionality associated with that option should still be executed, before the menu is presented again. Think of how you could use a variable to achieve this.

The user, at this point, should be able to enter any numeric value. For example, the user could enter a value of **42** to the menu. If the user does enter a value that is invalid, output an error message informing them as such, then ask the user to enter another value.

You will first need to think about which loop structure is the most appropriate. Then, adjust your pseudocode such that the above change is reflected. Then, edit your Java code such that it implements the changes you've made to your pseudocode. Finally, create an appropriate test plan and use it to test that your program works as intended.

2. Using loops for user input

Another use for loops is to validate user input, prompting the user to enter input repeatedly until it is deemed 'valid'. User input validation – also known as sanitising your input – is crucial for any program, as it stops the program attempting to use invalid data. This could lead to security issues or the program not functioning as intended. To do this, we assume the user doesn't know what they are doing and that we cannot guarantee that if we ask them to enter a valid value, that they actually will do it accurately.

Below is an example of pseudocode for generic user input that uses looping to ensure valid input. It is expected as PDI students that you will handle input in this way or similar throughout your work for the unit.

```
minValue <- 0 // Whatever your minimum valid value is.
maxValue <- 10 // Whatever your maximum valid value is.

DO:
    enteredValue <- ASK USER WITH PROMPT "Please enter a value: "
    IF enteredValue > maxValue OR enteredValue < minValue:
        PRINT "Error: the value must be between " minValue " and " maxValue
    END IF
WHILE enteredValue > maxValue OR enteredValue < minValue
```

The above pseudocode describes a solution that prompts the user and stores their input in a variable named **enteredValue**. If the user enters a valid value - between **minValue** and **maxValue** exclusive, then the **IF** statement won't be triggered and no error message will be output. As a **DO-WHILE** loop has its condition checked at the end of the loop, a valid value will mean that the condition is **False** and execution will continue below the loop.

In the case that the input is invalid – that is, the **enteredValue** is greater than the **maxValue** or smaller than the **minValue** – then the **IF** statement will be triggered, an error message output and then the loop would be run again as the **WHILE** condition would be **True** and execution returns to the top of the loop.

A **DO-WHILE** loop is used as the code needs to run **one or more times**. The user must be prompted at least once for their input.

> **Note:** Ensure that you set **minValue** and **maxValue** to appropriate values for the problem you are solving. The values set above are merely placeholders.

Your task is to further adjust your **Menu** pseudocode such that the user is continually prompted for input to select a menu option until they enter a valid value – that is, an integer number corresponding to a menu option, including the 'Exit' option. At this point, you do not need to consider the situation that a user would enter a non-integer input, such that you can continue to use **sc.nextInt()** in the same manner you are used to.

Once you have adjusted your pseudocode, implement the changes in your Java code. Run your program and update your pseudocode and Java code to reflect any changes required due to errors discovered during testing.

3. Non-integer input validation

Once you are comfortable with having validated the menu choice options in the question above, create a copy of your **CelsAndFaren** pseudocode and Java code from **P03**.

Your task is to adjust your pseudocode design such that the user is continually prompted to enter a value for which type of conversion they would like to be undertaken until they have entered a valid choice. In other words, user input for the 'choice' of conversion must be repeated until either **'c'**, **'C'**, **'f'** or **'F'** has been input. You will need to modify the conditions described in the question above for to handle this type of input.

> **Note:** Think of how you could utilise the **ELSE** and **DEFAULT** in your **IF** and **CASE** statements respectively to minimise redundancy in your code.

Once you have adjusted your pseudocode, implement the changes in the corresponding Java code file. Run your program and update your pseudocode (and Java code) to rectify any errors encountered in the process of doing so.

4. Fixed-iteration loops

The above exercises have demonstrated how we can use **DO-WHILE** and **WHILE** loops to repeat code a number of times which is determined by whether a boolean equation is **true** or **false**. However, there are often occasions where we wish to repeat a piece of code a specific number of times. That number may not be fixed (i.e. it is a variable set from user input), however it does not vary within the loop. This is where it is appropriate to use a **FOR** loop that executes a specified number of times. It is possible to use one of the other types of loops for this purpose, however it is more compact and easier to read a **FOR** loop that achieves the same purpose – you are less likely to create a logic error during the process.

Using pseudocode, design a program which asks the user to enter an integer number. Then, output the integer numbers between 1 and the input number inclusive, separated by a space, on the same line. It is OK if the line wraps onto the next line for large numbers – this is unavoidable! For example, if the user was to enter 3, the output would look similar to the following:

```
Enter an integer number: 3
1 2 3
```

Once you have designed your program using pseudocode, implement your program using Java. Run your program, and fix any errors your encounter while doing so.

5. One-dimensional arrays

You are most likely aware of arrays from the lectures; it is now time to take a look at howe they can be used in practice!

The variables that we have seen so far represent a single item of a single data type, but we also often work with sets or groups of items of a similar data type – for example, a set of student marks. Rather than having each mark stored in a singular variable, we can store them

all in the same variable – an array! Arrays are the simplest kind of data structure for storing a set of data. Instead of just one item (or **element**), an array is a variable that contains many elements of the **same** data type.

The variable itself is a reference to the first element of the array. In Java, the "array variable" also knows how large the array is – meaning you must declare the length of the array – the number of elements it will contain – when you declare the array itself. This is because the array is stored in a contiguous block of memory, so Java needs to know how much memory to allocate.

Your task is to design, in pseudocode, an algorithm that will:

- Ask the user to enter a whole number representing the length of an array to allocate. This number must be between **1** and **20** inclusive.
- Assign a random number to every element of the array, between **1** and the length of the array (inclusive). A description of how to do this is provided below.
- For each element of the array, output to the screen in the following format, starting at **1**: **"The value of element 1 of the array is 7."**, where the first number is the element of the array (starting at **1** and the second number is the random value assigned to it.

Note: Random should be used in a similar manner to **Scanner**. Below is an example of how to use **Random** in Java:

```
Random rand = new Random();
int randomNumber, maxValue;

maxValue = 20; // 'Maximum' random number generated.

randomNumber = rand.nextInt(maxValue);
```

So, just as you declare a **Scanner** at the top of your **main**, you will need to do the same for **Random**. Only the last line needs to be repeated each time you wish to generate a random number. Ensure that you read the Java APIdocumentation such that your call to **nextInt** generates a random number within the required boundaries. It is not as simple as it seems!

6. Errors, redundancies and inefficiencies

The algorithm below contains at least three errors, inefficiencies or redundancies. Describe at least three of them, ensuring you state **why** each of them is a problem.

- **What is an error?** An **error** is a piece of code that is incorrect, or has incorrect logic or does not align with the coding standards.
- **What is a redundancy?** A **redundancy** is a piece of code that 'does nothing', meaning there is no point having it in the code either because it has no functionality or it is repeating the functionality of something else.

- **What is an inefficiency?** An **inefficiency** is a piece of code that *could* be re-written in a different way to do the same job as the original piece of code, but better and more efficient.

> **Note:** The purpose of the algorithm is not relevant!

```
found = TRUE
numElements = 0

numElements (Integer) <- ASK USER WITH PROMPT "Please enter the length you'd like ↩
    your array to be: "

CREATE array OF INT, SIZE OF numElements // Assume the array is filled here.

searchInt (Integer) <- ASK USER WITH PROMPT "Please enter the integer you would ↩
    like to search for in the array:"

FOR i = 0 TO numElements INC BY 1
    foundAt = 0
    IF array[i] EQUALS numElements THEN
        foundAt = i
        found = TRUE
        BREAK
    END IF
END FOR

PRINT "Integer found at" foundAt
```

> **Note:** To get the most out of this activity, it is strongly advised you at least attempt to find at least one error in **each** category. Before you ask your tutor what an error, redundancy or efficiency is, please read the definitions above.

7. Exception Handling

Background

An **exception** occurs when your program enters an undesirable state and further execution is halted, because the program cannot continue in such a state.

We can quite often reach these states when considering user input, as users' won't always play nice and enter input in the format we desire. For example, our various calls to functions of **Scanner** can **throw** (or raise) exceptions when the user enters data in the wrong format. This can be seen in the Java documentation for the **Scanner** class (such as at this link).

We are generally (currently) interested in capturing the **InputMismatchException** that is raised when a call to **nextInt()** or **nextFloat()** is made. The exception is raised when a user enters

something that is not an integer or floating point number respectively; for example, they could enter **X** rather than **0** to exit a menu, such as the one described above.

> **Note:** Due to the way that **Scanner** handles user input in the case of **nextInt** and **nextFloat**, it is most advisable for your program to quit when incorrectly formatted input is provided to one of these functions.

Exceptions are generally handled using a **try-catch** block. The code that is to be attempted to be executed is specified within the **try** block. The Java Virtual Machine will interpret this as normal, from top to bottom. If an exception is thrown, execution of this block will stop, the **catch** block will be executed and then the rest of the program will be executed as per normal. If exceptions are not caught, however, overall execution of the program will cease.

The Task

Your task is to modify your pseudocode for **Menu**, such that **InputMismatchException**s are handled for when users provide input. It is entirely permissible to inform the user they have entered incorrect data and then continue to loop to ask for the input again.

Once you have updated your pseudocode, implement your changes using Java. Then, compile and run your program, updating your pseudocode and Java code with any errors you identify through testing your program.

End of Worksheet