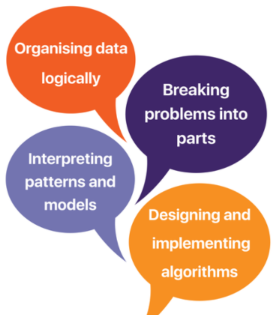
What is computational thinking?

-Logical and critical thinking, problem solving.
-Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer -human or machine- can effectively carry out.
-The mental activity for abstracting problems and formulating solutions that can be automated.
-The process of recognizing aspects of computation in the world that surrounds us, and applying tools and techniques from Computer Science to understand and reason about both natural and artificial systems and processes. (Furber, 2012)
-A mental orientation to formulating problems as conversions of some input to output and looking for algorithms to perform the conversions. Today the term has been expanded to include thinking with many levels of abstractions, use of mathematics to develop algorithms, and examining how well a solution scales across different sizes of problems.

Pillars of Computational Thinking:


Organising data logically
Breaking problems into parts
Interpreting patterns and models
Designing and implementing algorithms

Decomposition: Breaking down a complex problem or system into smaller parts that are more manageable and easier to understand. The smaller parts can then be examined and solved, or designed individually.
Pattern Recognition: When we decompose a complex problem we often find patterns among the smaller problems we create. The patterns are similarities or characteristics that some of the problems share.
Abstraction: Focusing on important parts only, ignoring irrelevant details. Filtering out the characteristics of patterns that we don't need in order to concentrate on those that we do. It is also the filtering out of specific details.
Algorithm Design: Developing the step by step instructions for solving this and similar problems. A sequence of clearly defined steps that describe a process to follow a finite set of unambiguous instructions with clear start and endpoints. There are two main ways to represent an algorithm: Pseudocode and flowchart.

Pseudocode

Pseudocode is an informal high-level description of a computer program or algorithm. It is written in symbolic code which must be translated into a programming language before it can be executed. Using Pseudocode is similar to "writing in a programming language" and might look something like this:

```
OUTPUT 'What is your name?' #INPUT asks a question. OUTPUT prints a message on the screen.
INPUT user inputs his/her name
STORE the user's input in the name variable
OUTPUT 'Hello' + name
OUTPUT 'How old are you?'
INPUT user inputs his/her age
STORE the user's input in the age variable
IF age >= 70 THEN
    OUTPUT 'You are aged to perfection!'
ELSE
    OUTPUT 'You are a spring chicken’
```

Pseudocode makes creating programs easier. Programs can be complex and long; preparation is the key. To use pseudocode, all you do is write what you want your program to say in English. Pseudocode allows you to translate your statements into any language because there are no special commands and it is not standardized. Writing out programs before you code can enable you to better organize and see where you may have left out needed parts in your programs. All you have to do is write it out in your own words in short statements.

```
Begin
INPUT hours
INPUT rate
pay = hours * rate
OUTPUT pay
End
```

more complex example:

```
Begin
INPUT hours, rate
IF hours < 40
THEN
    pay = hours * rate
ELSE
    pay = 40 * rate + (hours – 40)
* rate *1.5
OUTPUT pay
End
```

How to Write Pseudocode - 1

- **Statements:** defined as an instruction that directs the computer to perform a specific action. In writing pseudocode, we will refer to singular instructions as statements. The order of execution of the statements is from top to bottom. This changes when using control structures, functions and exception handling.

  - **Mathematical Operations:** are integral to solution development. They allow us to manipulate the values we have stored. Here are common mathematical symbols:

```
Assignment: ← or :=
    Example: c ← 3, c := 2
Comparison: =, ≠, <, >, ≤, ≥
Arithmetic: +, −, ×, /, mod
Logical: and, or
```

  - **Keywords:** reserved by a program because the word has a special meaning. Keywords can be commands or parameters. Every programming language has its own keywords (reserved words). Keywords cannot be used as variable names. In Pseudocode, they are used to indicate common input-output and processing operations. They are written fully in uppercase.

```
START, BEGIN: This is the start of your pseudocode.
INPUT: This is data retrieved from the user through the input device.
READ, GET: This is used when reading data from a data file.
PRINT, DISPLAY, SHOW, OUTPUT: This will show your output to a screen.
COMPUTE, CALCULATE: To calculate the result of the expression.
SET, INIT: To initialize values
INCREMENT, BUMP: To increase the value of a variable
DECREMENT: To reduce the value of a variable
END: This is the end of your pseudocode
```

- **Conditionals:** During algorithm development, we need statements that evaluate expressions and execute instructions depending on whether the expression evaluated to True or False. Here are some common conditions used in Pseudocode:

  - **If - Else If - Else:** This is a conditional that is used to provide statements to be executed if a certain condition is met. This also applies to multiple conditions and different variables.

Here is an if statement with one condition:

```
IF you are happy
   THEN smile
ENDIF
```

Here is an if statement with an else section. Else allows for some statements to be executed if the "if" condition is not met.

```
IF you are happy
   THEN smile
ELSE
    frown
ENDIF
```

We can add additional conditions to execute different statements if met.

```
IF you are happy
   THEN smile
ELSE IF you are sad
    THEN frown
ELSE
    keep face plain
ENDIF
```

  - **Case:** Case structures are used if we want to compare a single variable against several conditions.

```
INPUT color
CASE color of
    red: PRINT "red"
    green: PRINT "green"
    blue: PRINT "blue"
OTHERS
#The OTHERS clause with its statement is optional.
   PRINT "Please enter a value color" #Conditions are normally numbers or characters.
ENDCASE
```

• Iteration:

To iterate is to repeat a set of instructions in order to generate a sequence of outcomes. We iterate so that we can achieve a certain goal.

- **FOR structure:** The FOR loop takes a group of elements and runs the code within the loop for each element.

```
FOR every month in a year
    Compute number of days
ENDFOR
```

- **WHILE structure:** Similar to the FOR loop, the WHILE loop is a way to repeat a block of code as long as a predefined condition remains true. Unlike the FOR loop, the WHILE loop evaluates based on how long the condition will remain true.

To avoid a scenario where our while loop runs infinitely, we add an operation to manipulate the value within each iteration. This can be through an increment, decrement, et cetera.

```
PRECONDITION: variable X is equal to 1
WHILE Population < Limit
    Compute Population as Population + Births – Deaths
ENDWHILE
```

• Functions:

When solving advanced tasks it is necessary to break down the concepts in a block of statements in different locations. This is especially true when the statements in question serve a particular purpose. To reuse this code, we create functions. We can then call these functions every-time we need them to run.

```
Function clear monitor
  Pass In: nothing
  Direct the operating system to clear the monitor
  Pass Out: nothing
Endfunction
```

-To emulate a function call in pseudocode, we can use the Call keyword:
`call: clear monitor`

• Program Wrapping:

After writing several functions in our pseudocode, we find the need to wrap everything into one container. This is to improve readability and make the execution flow easier to understand.
To do this, we wrap our code as a program. A program can be defined as a set of instructions that performs a specific task when executed.

```
PROGRAM makeacupoftea
```

```
END
```

• Conclusion:

There are no technical rules for Pseudocode. It is meant to be human-readable and still convey meaning and flow.
There are different guide and tutorials which lean more towards language-specific pseudocode, examples of such are Fortran style pseudocode, Pascal style pseudocode, C style pseudocode and Structured Basic style pseudocode.

• Exception Handling:

An exception is an event which occurs during program execution that disrupts the normal flow of the instructions. These are events that are non-desirable.
We need to observe such events and execute code-blocks in response to them. This is called exception handling.

```
BEGIN
    statements
EXCEPTION
    WHEN exception type
        statements to handle exception
    WHEN another exception type
        statements to handle exception
END
```

How to Write Pseudocode - 2

We're going to break down the process of writing pseudocode step by step so that you know exactly how to write it and how to use it effectively.

1. Understand the Uses:

For starters, pseudocode simply makes the task of creating a new computer program more simple and straightforward.

Writing out the code in English enables you to create a verbal outline to follow during the programming stages of the project.

Pseudocode gives you the tools needed to ensure that everything you need will be included during programming. It lets you catch mistakes before they become mistakes.

It's also highly beneficial to use pseudocode for group projects. It breaks the program down in a simple manner so that all programmers are on the same page.

2. Pseudocode is Subjective:

There is no standard way to write pseudocode. The goal is simply to properly outline everything in your mind.

That said, there are certain structures and standard procedures you should use if you're working with others. Follow these rules to ensure that everyone else on the team is on the same page.

Perhaps the most important rule is to place clarity first. Make your pseudocode as clear and concise as possible, so there is no question as to what you mean.

3. Algorithms and Basic Constructs:

Two of the most important things you must understand when it comes to writing pseudocode are algorithms and basic algorithm constructs.

- Understand Algorithms – An algorithm is the steps you must take to achieve a specific goal.
- Know Algorithm Flow – The most basic algorithm construct or flow is "sequence," "selection," and "iteration." These lay out the proper way to write the code.
- Combine the Pieces – Take the information you want to relay and use algorithm flow to create a straightforward outline.

4. Standard Procedure:

As mentioned above, there is no standard procedure for writing pseudocode. However, that doesn't mean there aren't certain rules you should follow.

Follow these basic rules to ensure that everyone you're collaborating with understands your pseudocode.

- One Statement Per Line – Express each statement or action on its own line.
- Capitalize Directions – Capitalize directions to highlight their importance (for example, "READ").
- Focus on Meaning – Write what the program will do. Don't write how to program it.
- Standard Programming Structures – Follow the algorithm flow discussed above to create easy-to-follow structures.
- Utilize Blocks – Group similar actions together into blocks to separate the pseudocode into separate steps.

5. Important Tips

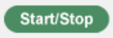
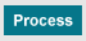
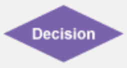
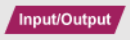
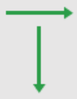
Once again, while pseudocode doesn't have any hard and fast rules, there are certainly some things you should do to make understanding the pseudocode easier for all involved.

- Keep It Simple – Simplicity and clarity are key. Write down what the actions will be, not how to program them.
- Explain Everything – Don't include information without explaining it. Add comments to explain your steps and reasoning if needed.
- Practice Makes Perfect – Just like learning a new programming language, learning how to write pseudocode takes time. Practice writing and reviewing it now.
- Review the Pseudocode – The biggest reason to write pseudocode is to catch any mistakes before programming. So, review the finished product thoroughly to nip errors in the bud.
- Translate into Programming Language – Implement the pseudocode by tracing it with your computer language. Compare the finished product to the pseudocode.

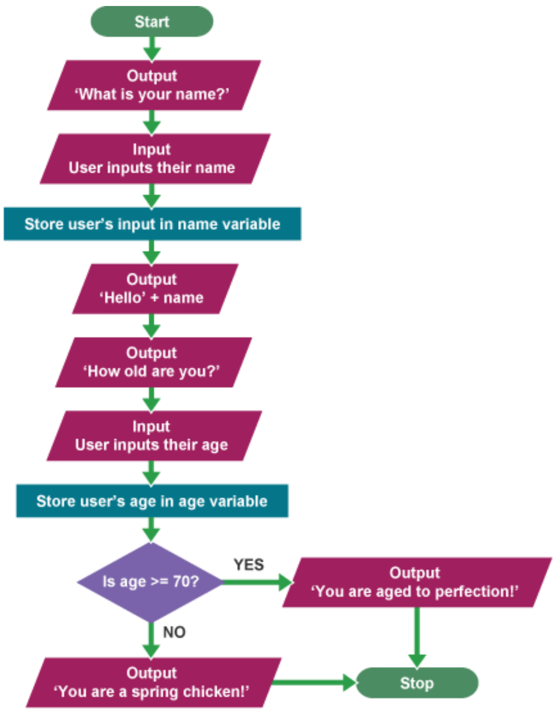
Flowchart

A flowchart is a diagram that represents a set of instructions. Flowcharts normally use standard symbols to represent different instructions. There are few real rules about the level of detail needed in a flowchart. Sometimes flowcharts are broken down into many steps to provide a lot of detail about exactly what is happening. Sometimes they are simplified so that a number of steps occur in just one step.

Flowchart symbols

Name	Symbol	Usage
Start or Stop		The beginning and end points in the sequence.
Process		An instruction or a command.
Decision		A decision, either yes or no.
Input or Output		An input is data received by a computer. An output is a signal or data sent from a computer.
Connector		A jump from one point in the sequence to another.
Direction of flow		Connects the symbols. The arrow shows the direction of flow of instructions.

A simple program could be created to ask someone their name and age, and to make a comment based on these. This program represented as a flowchart would look like this:


Start
Output 'What is your name?'
Input User inputs their name
Store user's input in name variable
Output 'Hello' + name
Output 'How old are you?'
Input User inputs their age
Store user's age in age variable
Is age >= 70?
YES
Output 'You are aged to perfection!'
NO
Output 'You are a spring chicken!'
Stop