Computational Thinking

Overview

Computational thinking is a flexible model programmers use to formulate a computer-based solution. One of the objectives of the computational thinking model is to get you **thinking like a programmer**, but also to aid in organizing your approach to problem solving. There are variations of this model, but commonly there are seven identifiable core parts:

1. Understand the Problem
2. Decomposition
3. Data Representation
4. Pattern Recognition
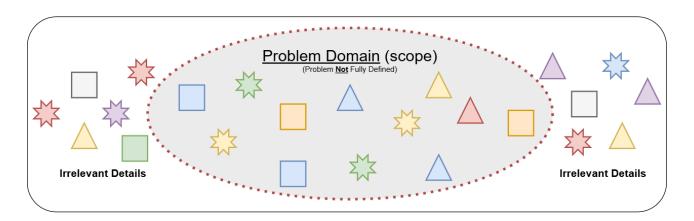5. Abstraction
6. Algorithm
7. Testing

The sequence of this list is based on the most common application of these parts, but by no means is it strict or concrete – depending on the problem, you may omit parts all together or swap some parts around as needed.

Problem solving is often iterative (repeating) and when changes are made in one major part it can have a cascading (snowball) effect on other parts. It is common to have to review other parts or at least anything related to parts that have changed, and this can be difficult to manage so you need to have a process that helps you do this which is where the computational thinking model comes in.

Ultimately what you want to accomplish is the creation of a complete correctly working **algorithm** that solves the problem which can be **used by a programmer to quickly and efficiently code the solution**.

Let's have a closer look at each of these core parts.

Understand the Problem

This may seem obvious, but understanding the problem is critical given the major side effects of what happens when the actual problem is misunderstood. If the problem is not fully understood, the outcome will almost certainly not be a solution to the problem resulting in many angry people (ranging from your peers, stakeholders, and even your family!).

Developing computer solutions is a time demanding and costly process so be sure to have a complete understanding of the problem to avoid wasting time and money on efforts not applicable to the problem you need to solve.

The most common failure is making assumptions. The problem is not always clear and depending on who is responsible for defining the problem (the client versus a project manager for instance), the details can be misleading or completely missing critical information. Assumptions are easy to make in cases where meaning is ambiguous or not explicit so be sure to confirm your

assumptions before proceeding.

Here is an example of a poorly worded problem that can lead to all sorts of possible "solutions" because it is too vague and ambiguous:

> "*The yearly revenue report doesn't work and needs to be corrected before we move on to the next phase of the project.*"

The issue is in the term "**doesn't work**". This phrase can so easily be misinterpreted - what exactly doesn't work?

- Are the calculations incorrect?
- Is the data corrupt?
- Could it be a minor cosmetic or formatting problem that only one person doesn't like?
- Is the application interface components not arranged in the way this one person likes?

If you make the assumption of any one of these things, you will most likely be incorrect. In these situations, you MUST seek clarity and precisely determine what "doesn't work" actually means. Without an explicitly defined problem, you can't deliver a solution!
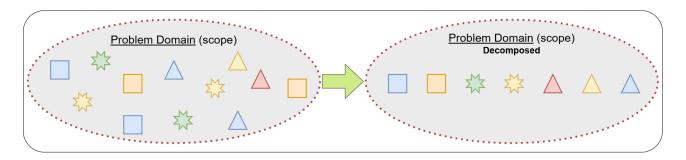
> ⚠️ **CAUTION**
>
> The viewpoint of a "problem" is often very subjective - what one person may see as an issue, another person may not agree. These issues are usually worked out by a project manager but for smaller projects, it is unlikely to have this benefit, so keep your guard up and always make sure there is general consensus supporting a common request.

Having a clear understanding of the problem will provide you with the **scope**

and boundaries of both the problem and the solution you need to create. This is extremely important as it will keep you focused on only the pertinent details of the problem and avoid wasting time, money, and effort into unrelated matters.

Decomposition

Most problems are too complex or are too dynamic in nature to immediately start creating a solution. Decomposing a problem into many smaller scoped problems greatly simplify many aspects of creating a solution. This step is mainly focused on identifying the major pieces of logic that can be extracted from the problem.

Isolating a specific part of a problem, removes irrelevant parts and greatly reduces the overall complexity for that part. This allows us to easily concentrate only on the important aspects of the smaller problem. Generally, there is no such thing as a complex problem – we just need to break it down into easier to solve smaller parts!

Here is an example of decomposition based on part of a much larger problem :

> "*The website should provide our administrators the ability to view the clerks who are currently logged-in to the system and for any selected clerk, provide options to send a message, disconnect them from the*

> *system, or assign a new task.*"

There are several significant pieces of the problem we can extract from this:

- The website user-interface (displaying the logged-in clerks and providing action options when a specific clerk is selected).
  - Possible function name: "**InitializeClerksLoggedIn**"
  - This function's scope will be limited to focusing only on preparing the interface, getting and displaying the logged-in user data listing, and providing the options for a selection.
- Sending a message to a specific clerk
  - Possible function name: "**SendClerkMessage**"
  - This function's scope will be limited to focusing only on how an administrator can send a message to a selected clerk from the list.
- Disconnecting a specific clerk
  - Possible function name: "**DisconnectClerk**"
  - This function's scope will be limited to focusing only on how a selected logged-in user is disconnected.
- Assigning a new task to a specific clerk
  - Possible function name: "**AssignClerkTask**"
  - This function's scope will be limited to focusing only on how a selected logged-in user is assigned a new task.

These are four major pieces of the problem that can be extracted and focused on to solve individually. These can be compartmentalized into specific functions where the logic can be isolated to solve only that specific problem.

The process of identifying smaller parts of the problem can **validate your understanding of the problem and confirms the overall scope**. Often this process will identify missed or undefined parts of the problem that will

need to be clarified which could likely expand the scope or sometimes have the opposite effect where irrelevant parts are identified and could likely reduce the scope.

Reducing a large problem into many smaller parts, **promotes a lot of flexibility in how you will orchestrate and reassemble these smaller solutions together when finalizing a complete solution**. It is important to take your time in this phase to filter for only the critical information and processes.

> ⚠ **NOTE**
>
> - This stage often identifies the potential core `functions` (procedures) to be created and used in the solution.
> - Functions represent algorithms comprised of several logical steps which perform a specific task (this will be described in more detail later on)

Data Representation

Contact-Name Cell-Number Work-Cell Email-Address
Country Work-Phone
Zip-Code First-Name Apartment Postal-Code
Unit-Number Street-Name City Last-Name Work-Email
Street-Number Time-zone

Information (data) is a major part of a computer-based solution since the data can significantly impact how the solution works and what it must do with the data. How data is received, used, or output is not the focus in this step, rather **the objective is to identify WHAT the relevant data is and to ensure it is represented in a way it can be used in the solution**.

Data representation is accomplished by representing data with `variables` (the technical aspect of this is covered later). Variables are named placeholders which can be referred to within the solution to access specific information by name to refer to the value. For example, if we needed to manage data about a person's contact information, some key data would be:

Information	Variable	Value
Full Name	`name`	`Jiminy Cricket`
Email Address	`email`	`jcricket@domain.com`
Cell Number	`cell`	`(123) 123-1234`

The important information in this example is a person's name, email address, and cell phone number. These important pieces of data are mapped or represented by variables which can be named to anything you wish but should almost always apply a **self-documented identifier to clearly represent the data while not being too long**.
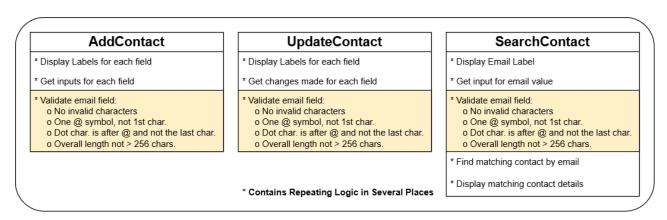
When we refer to the variable `name`, it will represent a value corresponding to that variable which in this example is `Jiminy Cricket`. Likewise, if we needed to refer to the email address data `jcricket@domain.com`, then the variable `email` would be used to target this information.

One question you may raise during this phase is about how much data you need to manage/represent and how granular (broken down into parts) you need to make it. In the preceding example, there is a `name` variable which includes the full name – does the solution require you to make a distinction between first and last name parts? If so, then you will need to represent this data separately where the `name` variable would need to be split into two

variables such as `firstName` and `lastName` (or surname). Eliminating the original variable `name`. When you need to refer to the full name, you would simply join these two variables accordingly.

There is an additional more advanced concept we can use to help manage more complex data representation, but this will be discussed later.

Pattern Recognition

AddContact	UpdateContact	SearchContact
* Display Labels for each field	* Display Labels for each field	* Display Email Label
* Get inputs for each field	* Get changes made for each field	* Get input for email value

* Validate email field:
  o No invalid characters
  o One @ symbol, not 1st char.
  o Dot char. is after @ and not the last char.
  o Overall length not > 256 chars.
	* Validate email field:
	  o No invalid characters
	  o One @ symbol, not 1st char.
	  o Dot char. is after @ and not the last char.
	  o Overall length not > 256 chars.
		* Validate email field:
		  o No invalid characters
		  o One @ symbol, not 1st char.
		  o Dot char. is after @ and not the last char.
		  o Overall length not > 256 chars.

		* Find matching contact by email
		* Display matching contact details

* **Contains Repeating Logic in Several Places**

After decomposing a problem into several smaller parts (let's say for example we have identified three functions as illustrated in the above image) and in outlining each of those functions, you notice repetition in the logic. The repetition can be within the same function and/or in other functions – this is undesirable! Why? Let's analyze the example further.

> ⚠ **NOTE**
>
> The breakdown of these functions as described below is an **overview only**. You will learn how to properly document logic later.

Function: `AddContact`

- This will detail all the steps needed to add the details of a new contact

- After the user enters an email address, the email field will be validated to ensure the email entered matches the expected format (`value@value.value`).
  - The email validation logic will require **_several steps of logic to implement_**:
    - Makes sure there are no invalid characters
    - Makes sure there is a single `@` symbol and not the first character
    - Makes sure there is a `.` symbol and at least 2 characters after the `@` and not the last character
    - Makes sure the overall length is not excessive (ie. > 256 characters)

Function: `UpdateContact`

- This will detail all the steps needed to update the details of an existing contact
- After the user enters an email address, the email field will be validated to ensure the email entered matches the expected format (`value@value.value`).
  - The email validation logic will require **_several steps of logic to implement_**:
    - Makes sure there are no invalid characters
    - Makes sure there is a single `@` symbol and not the first character
    - Makes sure there is a `.` symbol and at least 2 characters after the `@` and not the last character
    - Makes sure the overall length is not excessive (ie. > 256 characters)

Function: `SearchContact` (based on email field)

- This will detail all the steps needed to search for a specific contact based on a user-entered email address and then display the contact details
- After the user enters an email address, the email field will be validated to ensure the email entered matches the expected format (`value@value.value`).
  - The email validation logic will require *several steps of logic to implement*:
    - Makes sure there are no invalid characters
    - Makes sure there is a single `@` symbol and not the first character
    - Makes sure there is a `.` symbol and at least 2 characters after the `@` and not the last character
    - Makes sure the overall length is not excessive (ie. > 256 characters)

All three functions contain the **same email validation logic** which means this would be coded (by a programmer) **three separate times!** Why is this undesirable?

What if the email validation logic has a bug (an error that doesn't properly validate the email)? You would have to review and change potentially all three functions that repeat this logic. It is very inefficient to maintain this type of design and is error-prone given the redundancy (could have three versions of validation all of which could be slightly different). Another case is if the email formatting rules change, again, this would require updating all occurrences of

this logic. What is the solution to this?

You should **extract the repeating logic into its own function** where the logic can be defined **once**, then in other parts of the solution where you need that logic implemented, you execute that logic when and where you need it. The function you create should be given a meaningful name that best describes what it does so it's easy to use and communicate. In this example the function could be called `ValidateEmail`. Now, the three other functions can be updated to CALL the new function `ValidateEmail` eliminating the redundancy of fully detailing how that logic works in three different places!

NEW **Function:** `ValidateEmail`

- Validates the email address to ensure the email entered matches the expected format (`value@value.value`) snf will be the **ONLY place where this logic is defined.**
- Makes sure there are no invalid characters
- Makes sure there is a single `@` symbol and not the first character
- Makes sure there is a `.` symbol and at least 2 characters after the `@` and not the last character
- Makes sure the overall length is not excessive (ie. > 256 characters)
- **Now this logic can easily be used by other parts of the solution whenever needed without repetition.**

Function: `AddContact`

- This will detail all the steps needed to add the details of a new contact
- After the user enters an email address, **CALL** `ValidateEmail`
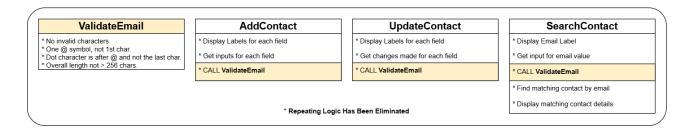

Function: `UpdateContact`

- This will detail all the steps needed to update the details of an existing contact
- After the user enters an email address, **CALL** `ValidateEmail`

Function: `SearchContact` (based on email field)

- This will detail all the steps needed to search for a specific contact based on a user-entered email address and then display the contact details
- After the user enters an email address, **CALL** `ValidateEmail`

Several lines of logic are now removed from each of the functions (`AddContact`, `UpdateContact`, and `SearchContact`) because there is now a common function called `ValidateEmail` and each function that needs this logic implementation simply **CALL**'s that function (ex: **CALL** `ValidateEmail`) when needed.

The below image illustrates the correction to the illustration at the beginning of this section.

ValidateEmail	AddContact	UpdateContact	SearchContact
* No invalid characters * One @ symbol, not 1st char. * Dot character is after @ and not the last char. * Overall length not > 256 chars.	* Display Labels for each field * Get inputs for each field	* Display Labels for each field * Get changes made for each field	* Display Email Label * Get input for email value
	* CALL **ValidateEmail**	* CALL **ValidateEmail**	* CALL **ValidateEmail**
			* Find matching contact by email
			* Display matching contact details

* Repeating Logic Has Been Eliminated

There is another situation where you may reveal a repeating pattern but does not necessarily require a function to address it. Sometimes the repetition is purely semantic (logical) and can be handled using **iteration** which is a

concept covered later.

> 💡 **SUMMARY**
>
> Patterns exhibiting a duplication in logic can usually be addressed by making a dedicated `function` which contains the logic in a single place. The function can then be called from anywhere in the solution to implement that logic whenever needed. Functions provide many benefits:
>
> - Single place to manage the logic
> - Simplifies the reading of logic (don't need to know how the logic works everywhere it is used)
> - Reduces the overall footprint size of the solution by eliminating redundant detailed logic
>
> There are also situations where logic duplication is purely semantic (logical) and can be managed using a logic control method referred to as `iteration`.

Abstraction

Abstraction is an extension of pattern recognition in that you can observe similar logic across many parts of the solution but are different in only minor ways, however the main essence of the logic is the same. Logic where the **overall concept or idea** can be reused but execute slightly differently across different contexts is an abstraction. Let's look at an example of this.

An application is needed to produce a business report for management that shows weekly data about their sales. The report data contains the **date** (formatted as: YYYY-MM-DD and is the Sunday of each week), **salesperson** name, and the total **sales** for that salesperson for that week - all reports use the same data but there are six main ways to view the report:

1. By the **date** -> ascending order
2. By the **date** -> descending order
3. By the salesperson's **total sales** -> ascending order
4. By the salesperson's **total sales** -> descending order
5. By the salesperson's **name** -> ascending order
6. By the salesperson's **name** -> descending order

> ⚠ **NOTE**
>
> Since all the report views use the same report data, we will represent the dataset with a **variable** `reportData`.

You might initially break this down so there is a function for each possible report view (notice the report data is sent to the function so the function can access the data):

1. `SortByDateDescending` (RECEIVES: **reportData**)
2. `SortByDateAscending` (RECEIVES: **reportData**)
3. `SortBySalesDescending` (RECEIVES: **reportData**)
4. `SortBySalesAscending` (RECEIVES: **reportData**)
5. `SortBySalesPersonAscending` (RECEIVES: **reportData**)
6. `SortBySalesPersonDescending` (RECEIVES: **reportData**)

If we want to view the report by date in ascending order, then we would have

to call the appropriate function:

```
CALL SortByDateAscending(reportData)
```

Each report view would have it's own specific function. This works fine, but you could further refine this by applying **abstraction** to reduce these 6 functions down to 3 functions. How? Each main report has two variants: sorting by **ascending** or **descending** order. We can **take the concept of sorting by ascending or descending and merge this logic into one function** for each main report view.

1. `SortByDate` (RECEIVES: **reportData**)
2. `SortBySales` (RECEIVES: **reportData**)
3. `SortBySalesPerson` (RECEIVES: **reportData**)

But wait - how can we use these functions to get the desired sorting order? Simple! We would send an additional variable or value to the respective function that would instruct the function the desired sort order. Let's update the functions now to include the extra information:

1. `SortByDate` (RECEIVES: **reportData**, **sortOrder**)
2. `SortBySales` (RECEIVES: **reportData**, **sortOrder**)
3. `SortBySalesPerson` (RECEIVES: **reportData**, **sortOrder**)

Now, when we call a report function we can send both the `reportData` **AND** include another piece of information that specifies the desired `sortOrder`. The function will use the `sortOrder` variable to determine the desired sorting order. Let's try it again now with this new abstracted function for the SortByDate:

```
CALL SortByDate(reportData, "ASCENDING")
```

When we call the function providing the "ASCENDING" **value**, it is assigned to

the variable `sortOrder` which the function will use to evaluate what order to sort the report.

Are we done yet? Could this be abstracted even further? YES! The primary concept being implemented in these functions is to sort data by a specific field (attribute) of data. We can apply the same idea as we did for the sortOrder and add another piece of information to send the function which would specify the **field** or attribute of the data to sort on! If we added this extra piece of information, we can now reduce these three functions down to **ONE!** Here's what it would look like:

`SortReport` (RECEIVES: **reportData**, **field**, **sortOrder**)

With this function, it can be called in many different ways but no matter what, it will sort the data based on the field we want and in the ascending or descending order using the conditions we send to the function that are captured in the variables: `field` and `sortOrder`. If we want the report to be sorted by salesperson in descending order, we can call the function like this:

```
CALL SortReport(reportData, "salesperson", "DESCENDING")
```

> 💡 **SUMMARY**
>
> Abstraction is the process of simplifying something back to a general concept or idea.

> ⚠️ **CAUTION**
>
> Sometimes abstraction can be over-applied (just because you can abstract something doesn't necessarily mean you should). There are varying scales at which you should apply it and is always evaluated on a case-by-case basis. If the abstraction creates too much additional complexity then you

should reevaluate and find another approach.

In the above example, where the second level of abstraction was applied where we reduced three functions down to one function, might have taken it too far as that function is now a lot more complex. Perhaps leaving it at the first level of abstraction having three functions would be more suitable and manageable to maintain (but is certainly better than the original six functions!).

Algorithm

An algorithm is ultimately what we are trying to accomplish. **An algorithm is a complete set of instructions in the sequence they must occur to solve a problem**. Algorithms describe in detail how the logic works step-by-step. This is what would be provided to a programmer to code a solution.

The scope of an algorithm can be small or vast depending on the problem it is addressing. When we decompose larger problems into smaller ones, each smaller one (ex: function) has its own algorithm, but when we design the 'main' function that orchestrates the overall solution, it will tie together other algorithms (functions) as required to provide a full set of instructions for the entire solution.

Since algorithms detail a solution, they must be tested to ensure it in-fact provides a working solution to the entire problem and stays within the scope of the problem (see next sub-section on Testing).

To manage and communicate algorithms, programmers primarily use **pseudo code** as it is more efficient to work with and can provide more detail. However, **flowcharts** are used to provide a higher level view of the solution and is generally less detailed due to the complexity of the layout using graphical symbols. These forms of communicating logic will be covered in the next major section (Documenting Logic).

Testing

Testing **should be done repeatedly throughout all parts of the computational thinking model**. The scale at which you test will depend on what it is you need to test. Making changes to established logic for instance would be one such time to do a concentrated and focused test to ensure the changes didn't break anything or cause other unforeseen side-effects.

The more targeted and frequent your testing is, the better your solution will be. However in reality, we usually do not have enough time to do extensive detailed testing, so we must be efficient about how and what we test (something you get better at in time).

What we minimally should test are all the known major logic flows that must occur to solve the problem and as the application would be used by users the majority of the time. We accomplish this by creating "use-cases". These are the expected and common scenario's that would occur in the execution of the solution. Prioritizing the features and critical logic parts of the solution that are used the most by users will validate and ensure the solution is mostly bug-free and confirms it actually solves the problem.

Testing is where we often reveal weaknesses in the logic and where applications mostly fail, is in the unexpected things! After the core logic of the solution is tested, you would move on to more robust testing and include out of

the norm conditions. One way to target this is the ask "what if.." and run that scenario through your solution to see if it works as expected. Be warned, once you start looking for exceptions and asking "what if", this can take you well beyond what is "reasonable" to test so know when to stop and when you have reached the "obscure" that goes well-beyond normal exceptions.

Summary

In summary, programmers should plan a solution applying the computational thinking model BEFORE starting any coding. Planning ahead and having a framework to work from **accelerates your coding time and results in substantially fewer errors** since the difficult part of determining the logic and flow of the solution is already done. **As a programmer, the coding part should only involve the syntax and implementation of the logic based on a prepared plan**.

The extent to which you apply the computational model will vary depending on the problem. All parts of the model are important and will lead to great improvements in your skills to build solutions, however a few of the parts are mandatory and you should most definitely get in the habit of doing the following ALWAYS:

- Understand the problem
- Decomposition

The other parts can take time and repetition in applying the concepts before you get comfortable and more skilled at using them. Abstraction is probably the most challenging of them because often it can be over-applied (just because you can abstract something doesn't necessarily mean you should, and there are varying scales at which you can apply it).

The act of **coding a solution into a program should actually be the least time consuming part of a project**. If you find otherwise, it likely means you

aren't working from a planned solution, or the prepared logical plan is poorly done and needs more work.

It is always worth the time to fix the plan than it is to waste significant time debugging and rearranging your code after it's been coded!

Documenting Logic

Overview

The previous section on computational thinking focused on the computational thinking model and how it applies to the problem solving process. Now, lets focus on how we will **document and communicate logic**.

There are many possible ways to document and communicate algorithms, however these notes will be focused on two very popular methods:

1. Flowchart
2. Pseudo Code

> **⊙ NOTE**
>
> For the purposes of these notes, we'll be using flowcharts to communicate a simplified view of an algorithm and pseudo code for communicating the details of an algorithm.

It was stated in the introduction to these notes, we will not be concerned with any specific programming language syntax (language agnostic) and will be focused primarily on the logic.

In staying with this theme and to establish **consistency and clear program-language agnostic communication** of algorithms, a minimal set of **guidelines** will need to be applied for each of these methods.

Flowchart

Flowcharts are highly visual as they are constructed with graphical symbols to represent information and process flows (algorithms) in a more **simplified view**.

Flowcharts are much more challenging to construct for complex detailed algorithms given the intensive application of graphical components - it can sometimes take more time to arrange the graphical symbols than it did to create the entire algorithm!

Flowcharts are used primarily for two purposes:

1. **To illustrate technical algorithms to non-technical persons.**
   - Flowcharts are a great way to communicate algorithms and technical information to non-technical persons who are not programmers or those who only need to have an overview of the algorithm and not the lower-level details.
2. **For technical persons to view detailed algorithms from a different perspective**
   - Technical persons like programmers will often use flowcharting to view a smaller more specific part of an algorithm to better understand it from a different perspective and to better "visualize" it. This often sparks more creative ways to address logic.
   - **Note**: Flowcharts with detailed algorithms is outside the scope of these notes and will not be covered.

Given these notes will be focused on **purpose #1**, this will limit a solution to having only one flowchart providing a simplified outline of the **main algorithm** needed to solve a problem. The flowchart will be an abstraction of the overall solution providing enough information to non-technical persons of the intent

and major logic flows of the solution.

Graphical Symbols

Flowcharts are actually standardized by the International Organization for Standardization ISO and depending on the sector/field of discipline and type of information you are using flowcharts for, will have its own set of standardized symbols that should be used. Overall there is an extensive library of symbols representing very specific meaning and should be used when appropriate.
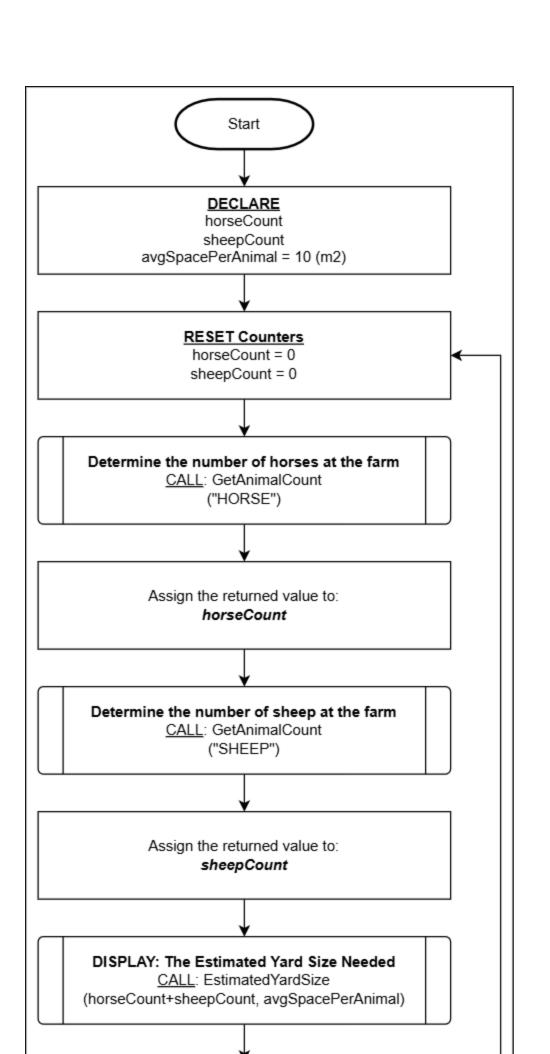
To stay within the scope of these notes, a very limited set of the most common symbols will be used:

Symbol-Image	Symbol-Shape	Description
Start	Squashed Oval	**Start** of the algorithm
End	Squashed Oval	**End** of the algorithm
Simple Process or Action	Rectangle	**Simple process** or action

Symbol-Image	Symbol-Shape	Description
Complex Function	Striped Rectangle	**Complex function** (a pseudo code function or a closed-box process)
Decision	Diamond	**Decision** (change in logical flow)
YES NO TRUE FALSE some other flow description	Text Label	**Text to describe decision output flow** (placed on top of the line)
→ ←	Lines with Arrows	**Flow direction** used to join symbols (indicates the next symbol to execute)

Simple Example

Throughout the notes as programming logic is introduced, detailed examples applying the guidelines will be provided. In the meantime, here's a simple example to give you some idea of what flowcharting is about.

```
                          ┌─────────────┐
                          │    Start    │
                          └──────┬──────┘
                                 │
                                 ▼
        ┌────────────────────────────────────────────────┐
        │                   DECLARE                        │
        │                  horseCount                      │
        │                  sheepCount                      │
        │           avgSpacePerAnimal = 10 (m2)            │
        └────────────────────────┬───────────────────────┘
                                 │
                                 ▼
        ┌────────────────────────────────────────────────┐
        │                RESET Counters                    │
        │                 horseCount = 0                   │
        │                 sheepCount = 0                   │
        └────────────────────────┬───────────────────────┘
                                 │
                                 ▼
        ┌─┐  ┌────────────────────────────────────┐  ┌─┐
        │ │  │ Determine the number of horses at   │  │ │
        │ │  │              the farm               │  │ │
        │ │  │      CALL: GetAnimalCount           │  │ │
        │ │  │            ("HORSE")                │  │ │
        └─┘  └────────────────────────────────────┘  └─┘
                                 │
                                 ▼
        ┌────────────────────────────────────────────────┐
        │          Assign the returned value to:           │
        │                  horseCount                      │
        └────────────────────────┬───────────────────────┘
                                 │
                                 ▼
        ┌─┐  ┌────────────────────────────────────┐  ┌─┐
        │ │  │ Determine the number of sheep at    │  │ │
        │ │  │              the farm               │  │ │
        │ │  │      CALL: GetAnimalCount           │  │ │
        │ │  │            ("SHEEP")                │  │ │
        └─┘  └────────────────────────────────────┘  └─┘
                                 │
                                 ▼
        ┌────────────────────────────────────────────────┐
        │          Assign the returned value to:           │
        │                  sheepCount                      │
        └────────────────────────┬───────────────────────┘
                                 │
                                 ▼
        ┌─┐  ┌────────────────────────────────────┐  ┌─┐
        │ │  │ DISPLAY: The Estimated Yard Size    │  │ │
        │ │  │              Needed                 │  │ │
        │ │  │     CALL: EstimatedYardSize         │  │ │
        │ │  │ (horseCount+sheepCount,             │  │ │
        │ │  │        avgSpacePerAnimal)           │  │ │
        └─┘  └────────────────────────────────────┘  └─┘
                                 │
                                 ▼
```

Pseudo Code

Pseudo code is a programmer's go-to choice for documenting algorithms. This is because of how flexible it is to modify and reorganize logic which is a very common activity when creating algorithms. Unlike flowcharts which is graphical, **pseudo code is textual and describes logic with concise clear human-language (English in this case)**. It is also easy to copy/paste the content directly into your code file to work from.

"Pseudo" meaning not genuine or false, indicates this type of describing logic is not genuine code (staying true to our rule of not programming in any specific language). Pseudo code can be more technical than flowcharts, but does not use any programming language specific syntax, but instead, **describes the logical intent at the level programmers can use to program the logic to any programming language desired**.

Pseudo code can be constructed in many different ways and generally we all have our own stylistic way of doing it but as mentioned in the overview, there will be some guidelines applied for the sake of consistency and clear communication. The guidelines described in these notes are minimal but effective general rules that are a good base to work from while you eventually establish your own style.

General Guidelines

Guideline	Description
Enumeration	Each step or instruction should have a corresponding sequence number or letter. When nested logic is applied,

Guideline	Description
	enumeration resets using the opposite number or letter sequence.
Indentation	The term nesting will be explained later, but it is crucial to consistently indent (or `TAB`) each level of logic that is nested.
Alignment	Overall horizontal left-alignment is critical to keeping statements organized and easily identifiable as to which each statement is related to or apart of other parts of logic (especially when nesting).
Flow-through Logic	Logic should flow naturally to the next step. You should not have instructions that state "continue to the next step" as this is the natural sequence when one statement ends, it automatically assumes to continue to the next statement in sequence.
Jump Statements	Logic flow needs to be carefully managed and NEVER apply "jump statements" which direct logic to jump ahead skipping many steps. Example: if step #5 instructs "GO TO step #99" this is extremely poor design and not acceptable!

Simple Example

As mentioned in the flowchart section, throughout the notes as programming logic is introduced, detailed examples applying the guidelines will be provided. In the meantime, here's a simple example to give you some idea of what

pseudo coding is about.

This example has two functions defined that were referenced from the preceding flowchart example above.

```
Function: GetAnimalCount(animalType)

1. DECLARE:
           count = 0

2. DISPLAY:
           Tap the [ADD 1] button to add 1 to the counter or
           [DONE] when you are finished counting

           <animalType> Count:<count>  BUTTON: [ADD 1]
                                       BUTTON: [DONE]

3. Which button was tapped?
     ADD 1:
           a) Add 1 to <count>
           b) Repeat from step:2

     DONE :
         a) DISPLAY:
                     Are you sure?
                     BUTTON: [YES]
                     BUTTON: [NO]
         b) Which button was tapped?
             NO:
                 1. Repeat from step:2

      NONE :
         a) Repeat from step:2

4. Return <count>
5. END
```

```
Function: EstimatedYardSize(totalAnimals, spacePerAnimal)

1. DECLARE:
        feedTrough = 3
        waterTrough = 6
        estAnimalSpace = <totalAnimals> * <spacePerAnimal>

2. DISPLAY:
        Do you need a feeding trough?
        BUTTON: [YES]
        BUTTON: [NO]

3. Which button was tapped?
    NO:
        a) feedTrough = 0

4. DISPLAY:
        Do you need a water trough?
        BUTTON: [YES]
        BUTTON: [NO]

5. Which button was tapped?
    NO:
        a) waterTrough = 0

6. DISPLAY:
        Total Animals    : <totalAnimals>
        Est. Animal Space: <estAnimalSpace> m2
        Feed Trough      : <feedTrough>
        water Trough     : <waterTrough>
        ----------------------------------------
        Total Estimate   : <estAnimalSpace> + <feedTrough> +
<waterTrough>

7. END
```

Variables

Overview

As introduced in the Computational Thinking: Data Representation section (Data Representation), variables are named placeholders which can be referred to within the solution to access specific information by name to refer to a value.

Variables are critical in providing our solutions the ability to not just store, but use information to help determine logical pathways (selection which is covered later). Information can be simple and straight forward, but it can also be very complex. The more complex data representation will be covered much later on in these notes, but in this section we'll concentrate on simple data representation and how variables are used in both flowchart and pseudo code methods of documentation.

Many programming languages are "typed" in that you must indicate the type of information the variable must represent (example: whole number, fractional number, characters etc.), but not all languages require this extra level of detail. Sticking with our "language agnostic" theme, we will not be including type information when we declare and use variables.

Terminology

Term	Meaning
IDENTIFIER	The **name** of the variable.

Term	Meaning
DECLARE	To **create** a variable which will include the IDENTIFIER.
ASSIGN	To **store a value** to a variable.
INITIALIZE	To both DECLARE and ASSIGN a value to a variable in one step.

Variable Identifiers (name)

Variable identifiers should be well though out and purposefully named to best describe the information it will be representing. It is important to carry this through into your actual programming as well! Providing meaningful names to your variables will contribute towards easier to read and manage logic (or code). However, we need to be mindful of efficiency as we don't want to be referring to very long names (and in the case of programming code, we don't want to be typing them either!), so we try to use a style of shorthand to shorten the names enough without losing the context.

For instance, if we need to represent information about a customer's primary phone number and an alternate phone number, we wouldn't want to be too descriptive:

```
DECLARE:
    customerPrimaryPhoneNumber
    customerAlternatePhoneNumber
```
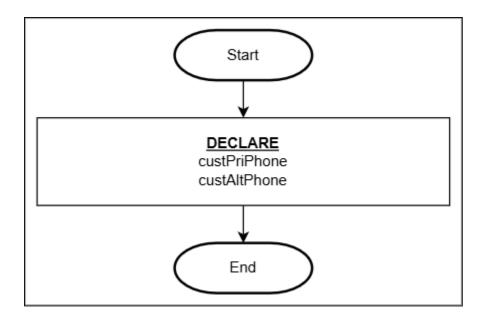
We would instead shorten the words to abbreviations but not to the point

where we still can't understand what they are:

Pseudo Code

```
1. DECLARE:
     custPriPhone
     custAltPhone

2. End
```

Flowchart

As you immerse yourself in this industry, you will continue to learn of typical shorthand naming conventions to represent typical types of information. You will be better in time and practice!

Something you may have noticed is the use of upper and lower case characters to help discern between words. This is a common naming practice ("lower camel case" - wikipedia) in programming which we will be applying in these notes as well.
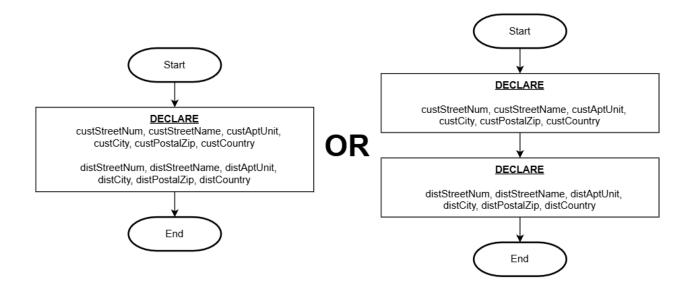
Organization

Sometimes we need to manage a lot of information which can lead to the use of many variables. It is a standard practice to group all the variables you will need using a combination of lines and line-spacing to help readers find variables and helps to see if you have covered all the data you will need representation for.

Scenario: A solution needs to represent address information for a customer and a distributor. How would you breakdown this information into appropriate variables?

Pseudo Code

```
1. DECLARE:
    custStreetNum, custStreetName, custAptUnit, custCity,
custPostalZip, custCountry
    distStreetNum, distStreetName, distAptUnit, distCity,
distPostalZip, distCountry

2. End
```

Flowchart

These variables are grouped by customer then by distributor and separating these two independent groups with a line break. Organizing the variables like this makes it super easy to find and confirm the data representation for each key piece of information.

🔥 DON'T DO THIS

What you don't want to do is declare every variable on a single line in random order! This is very confusing and benefits nobody:
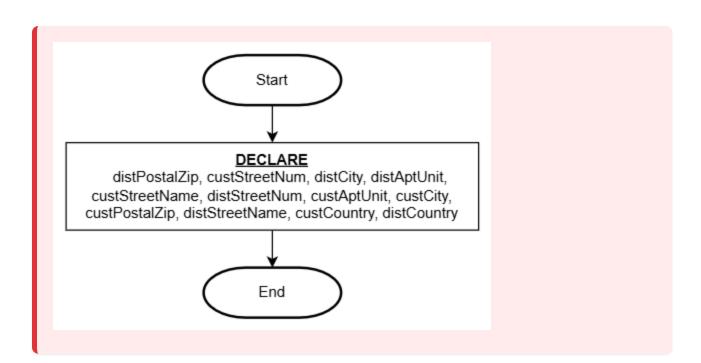
Pseudo Code

```
1. DECLARE:
    distPostalZip, custStreetNum, distCity, distAptUnit,
custStreetName, distStreetNum, custAptUnit, custCity,
custPostalZip, distStreetName, custCountry, distCountry

2. End
```

Flowchart

```
        ┌─────────────┐
        │    Start    │
        └──────┬──────┘
               │
               ▼
┌──────────────────────────────────────────────┐
│                  DECLARE                       │
│  distPostalZip, custStreetNum, distCity,      │
│  distAptUnit, custStreetName, distStreetNum,  │
│  custAptUnit, custCity, custPostalZip,        │
│  distStreetName, custCountry, distCountry     │
└──────────────────┬───────────────────────────┘
                   │
                   ▼
            ┌─────────────┐
            │     End     │
            └─────────────┘
```

Modularity with Functions

Overview

Introduce functions...

High-Level Functions

The concept of 'main' overall logic flow - no deep details use flowchart.

Low-Level Functions

Full details of a focused part of logic - pseudo code.

Passing Information

Explain how to document passing of data to a function.

Returning Information

Explain how to return data from a function

Explicit Return

Explain how to return explicitly - use return

Implicit Return

Explain how to return implicitly - via argument(s)

Typical Solution

A typical solution will contain a single flowchart describing the overall main parts of logic in the solution. It will contain references to functions where the full logic detail can be found which will be documented in pseudo code.

> 🔥 **TODO**
>
> Provide a simple example of this architecture

Logic - Selection

Overview

Selection is the concept representing decisions...

Basic Selection

> 🔥 **IMAGE**
>
> Show an image with straight vertical line and a bend going to the right (if).

Optional selection where something may happen, otherwise, continue execution...

- Flowchart
- Pseudo Code

Alternative Selection

> 🔥 **IMAGE**
>
> Show an image with a Y line showing a bend to both the left and right (if/else).

A 'Y' in the road - a decision must be made and no matter what one of two things will occur but not both.

- Flowchart

- Pseudo Code

Multiple Alternative Selection

> 🔥 **IMAGE**
>
> Show an image with a E line showing multiple logic flows (if, else | else-if).

A 'Y' in the road - a decision must be made and no matter what one of two things will occur but not both.

- Flowchart
- Pseudo Code

Nesting Selection

Do some examples

- Flowchart
- Pseudo Code

Iteration

Overview

Iteration provides logic that can be repeated as a loop...

Optional Iteration

May enter a loop (for/while)...

How it is done for both methods:

- Flowchart
- Pseudo code

Mandatory Iteration

Will loop at least once (do-while)...

How it is done for both methods:

- Flowchart
- Pseudo code

Nesting Iterations

Do some examples

- Flowchart

- Pseudo Code

Data Collections

Overview

The simplest form of a data collection is an array. The concept of an array is a variable that can store many items...

Declaring an Array Variable

How it is done for both methods:

- Flowchart
- Pseudo code

Accessing Data of an Array

- Specific item
- All items (iteration)

How it is done for both methods:

- Flowchart
- Pseudo code

Data Structures

Overview

Data structures...

Declaring a Data Structure

How it is done for both methods:

- Flowchart
- Pseudo code

Declaring a Data Structure Variable

How it is done for both methods:

- Flowchart
- Pseudo code

Accessing Data

- Specific member/attribute

How it is done for both methods:

- Flowchart

- Pseudo code

Declaring a collection of Data Structures (array)

How it is done for both methods:

- Flowchart
- Pseudo code

Accessing Data of an Array of Data Structures

- Specific item
- All items (iteration)

How it is done for both methods:

- Flowchart
- Pseudo code