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

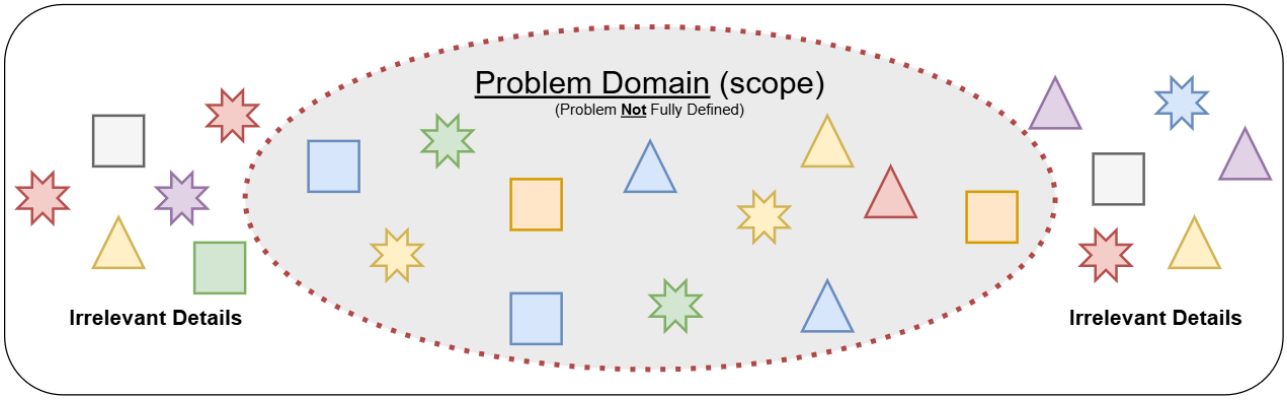
The sequence of this list is based on the most common application of these parts, but by no means is it strict or concrete – depending on the problem, you may omit parts all together or swap some parts around as needed.

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?

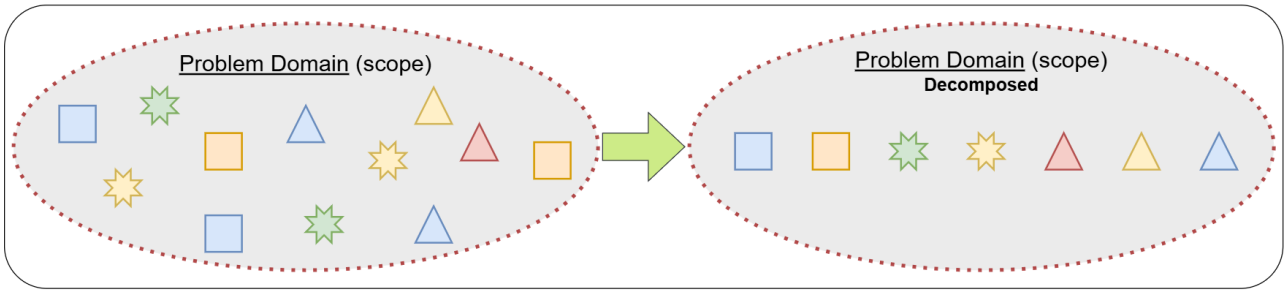
If you make the assumption of any one of these things, you will most likely be incorrect. In these situations, **you MUST seek clarity** and precisely determine what "doesn't work" actually means. Without an explicitly defined problem, you can't deliver a solution!

> **⚠ 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 part of the problem.

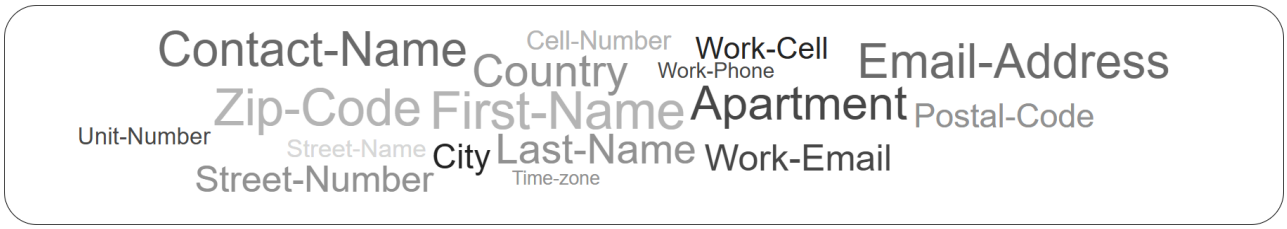
The process of identifying smaller parts of the problem can **validate your understanding of the problem and confirms the overall scope**. Often

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



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.

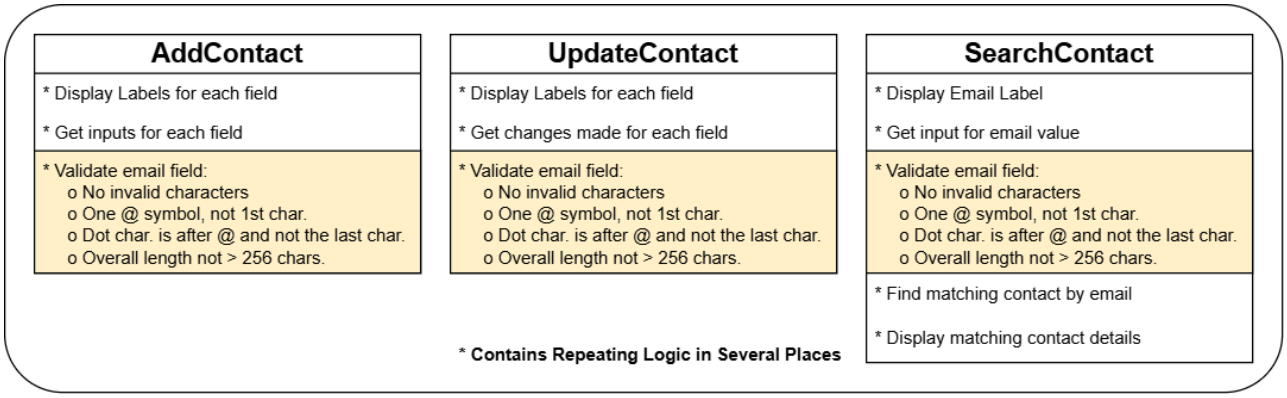
One question you may raise during this phase is about how much data you need to manage/represent and how granular (broken down into parts) you need to make it. In the preceding example, there is a `name` variable which includes the full name – does the solution require you to make a distinction

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 in two other sections:

- data collections
- data structures

Pattern Recognition



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.**

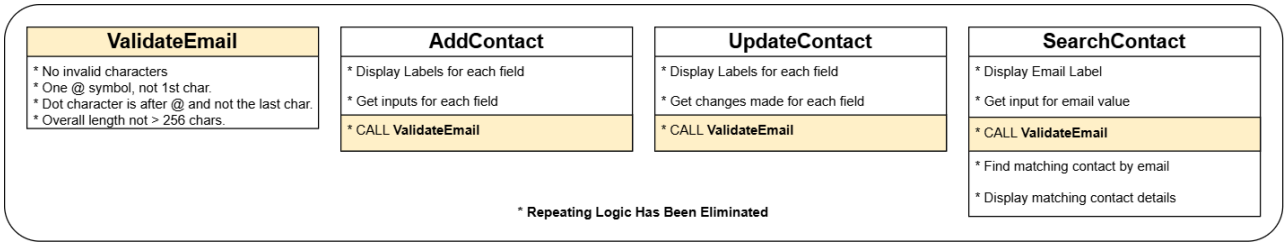
> **Function:** `AddContact`
>
> - This will detail all the steps needed to add the details of a new contact
> - After the user enters an email address, **CALL** `ValidateEmail`

> **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.



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.

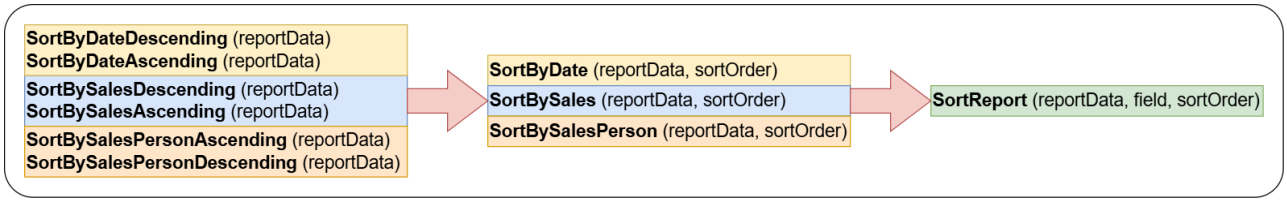
SUMMARY

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

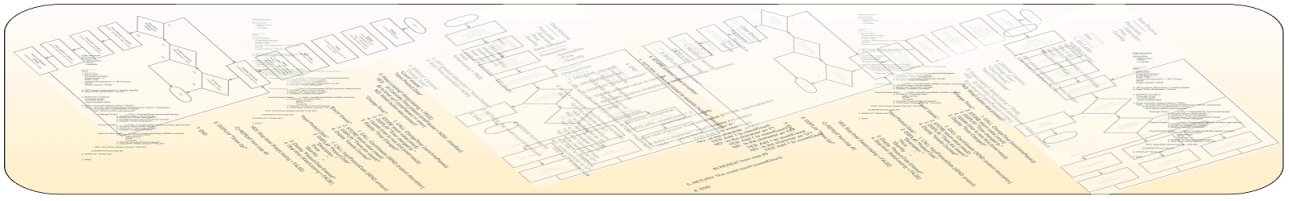
sorted by salesperson in descending order, we can call the function like this:

```
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 **pseudocode** 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 couple of the parts are mandatory and you should definitely get in the habit of doing the following ALWAYS:

- Understand the problem
- Decomposition

The other parts may 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. Pseudocode

> **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.

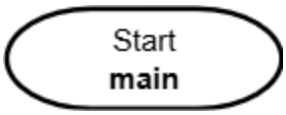
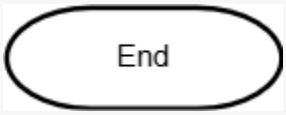
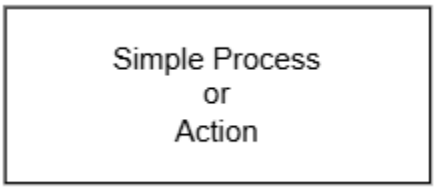
Given these notes will be focused on **purpose #1**, this will limit a solution to having only one flowchart providing a simplified outline of the **main algorithm** needed to solve a problem. The flowchart will be an abstraction of the overall solution providing enough information to non-technical persons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

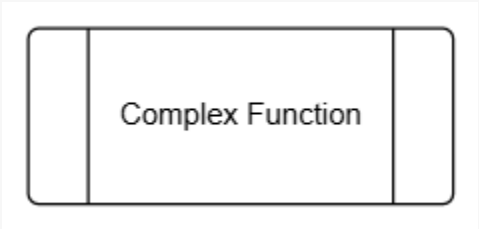
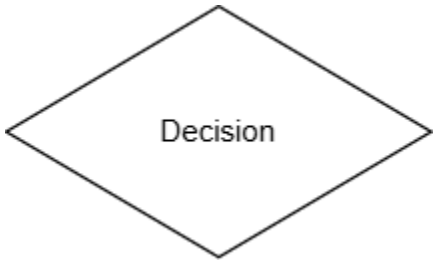
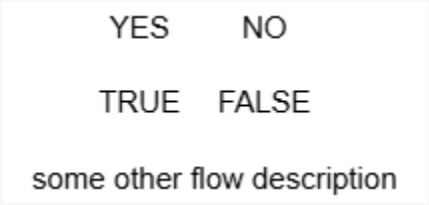
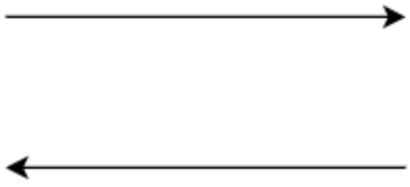
and major logic flows of the solution.

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.

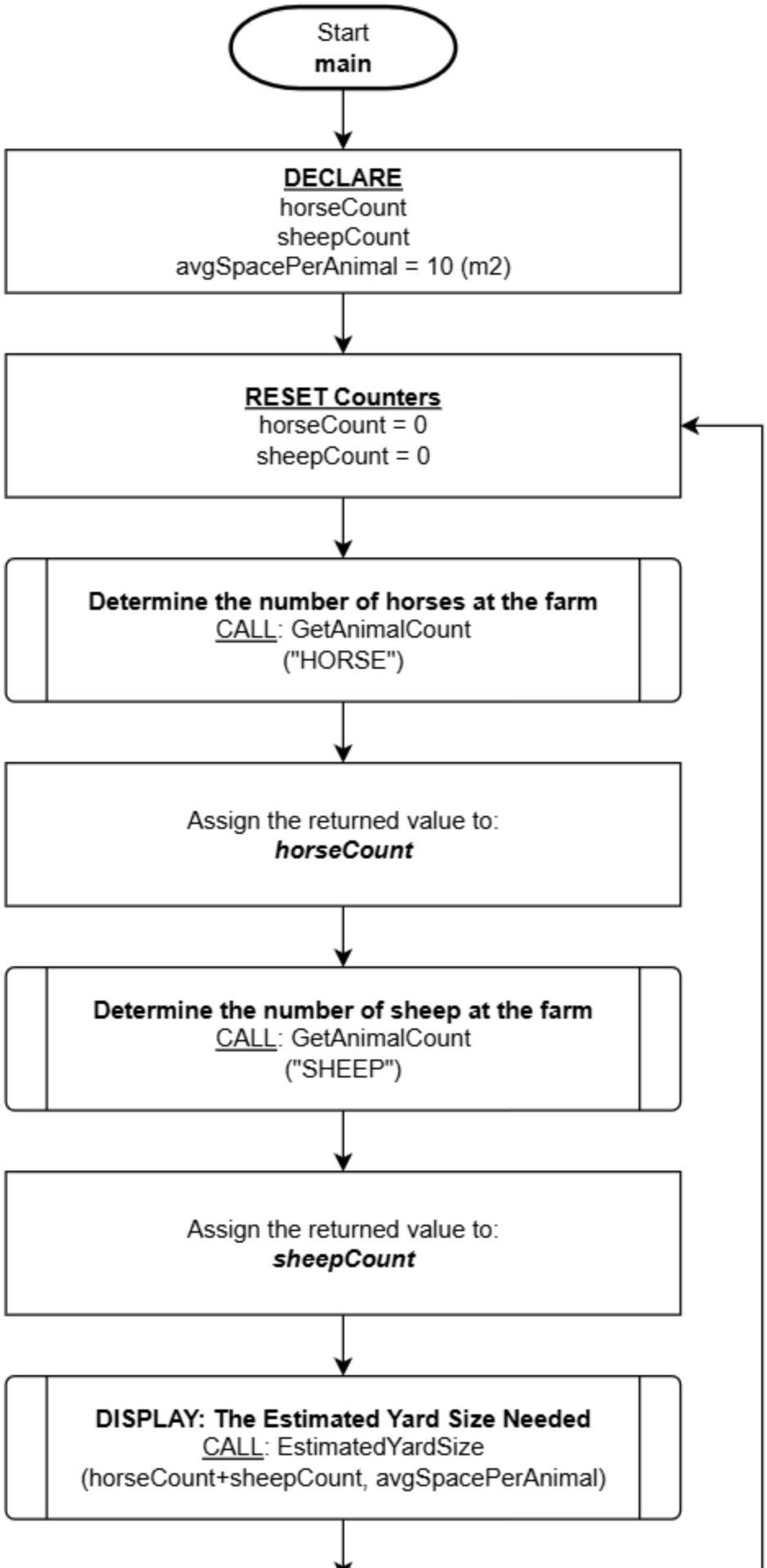
To stay within the scope of these notes, a very limited set of the most common symbols will be used:

Symbol-Image	Symbol-Shape	Description
	Squashed Oval	**Start** of the algorithm (with function name)
	Squashed Oval	**End** of the algorithm
	Rectangle	**Simple process** or action

Symbol-Image	Symbol-Shape	Description
	Striped Rectangle	**Complex function** (a pseudocode function or a closed-box process)
	Diamond	**Decision** (change in logical flow)
	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
main
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)

Pseudocode

Pseudocode 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, **pseudocode 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). Pseudocode 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**.

Pseudocode 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 and how 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.

```
------------------------------------------------------------------
Description/Purpose:
   Prompts the user to increment a counter for the animalType
   specified by given argument and returns the count value

Argument(s) : animalType (a string representing the animal type)

Return Value: count (# of animals counted: whole number)
------------------------------------------------------------------

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?
    A. ADD  :
            1. ASSIGN: count = count + 1 <---- Note: Add 1 to the
count
            2. REPEAT: from step 2

    B. DONE :
            1. DISPLAY:
                      "Are you sure?
```

```
-----------------------------------------------------------------
Description/Purpose:
   Calculates the estimated yard size required based on the
   received number of animals and average space used by a single
   animal. The resulting calculated area is returned. This also
   factors optional water trough and feeding trough options.

Argument(s) : totalAnimals (whole number)
              spacePerAnimal (average area required for a
                              single animal)

Return Value: area (calculated total area needed for all animals)
-----------------------------------------------------------------

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?
      A. NO:
            1. ASSIGN: feedTrough = 0

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

5. Which button was tapped?
      A. NO:
```

Variables

Overview

As introduced in the Computational Thinking: Data Representation section, 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 in these notes, but in this section we'll concentrate on simple data representation and how variables are used in both flowchart and pseudocode 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 the "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 thought out and purposefully named to best describe the information it will be representing. It is important to apply this to 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
```

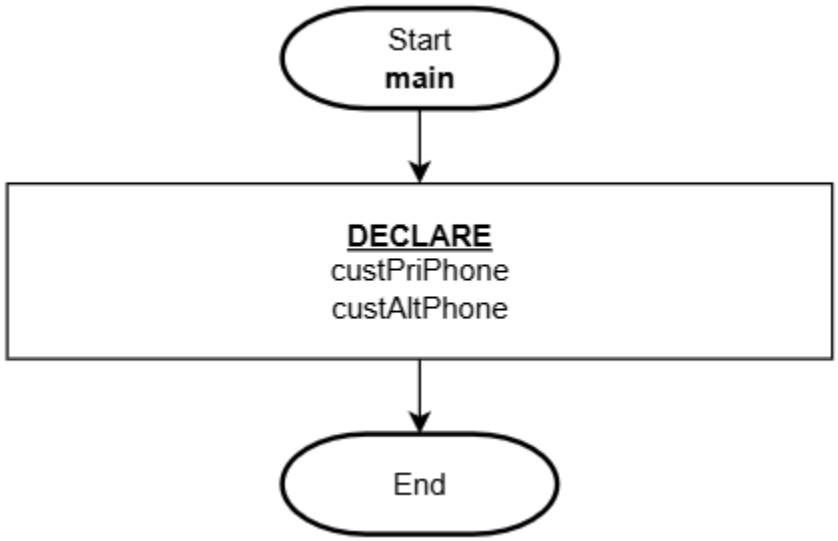
We would instead shorten the words to abbreviations but not to the point

where we still can't understand what they are:

Pseudocode

```
1. DECLARE:
    custPriPhone
    custAltPhone

2. End
```

Flowchart



As you immerse yourself in this industry, you will continue to learn many typical shorthand naming conventions for various information. You will get 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

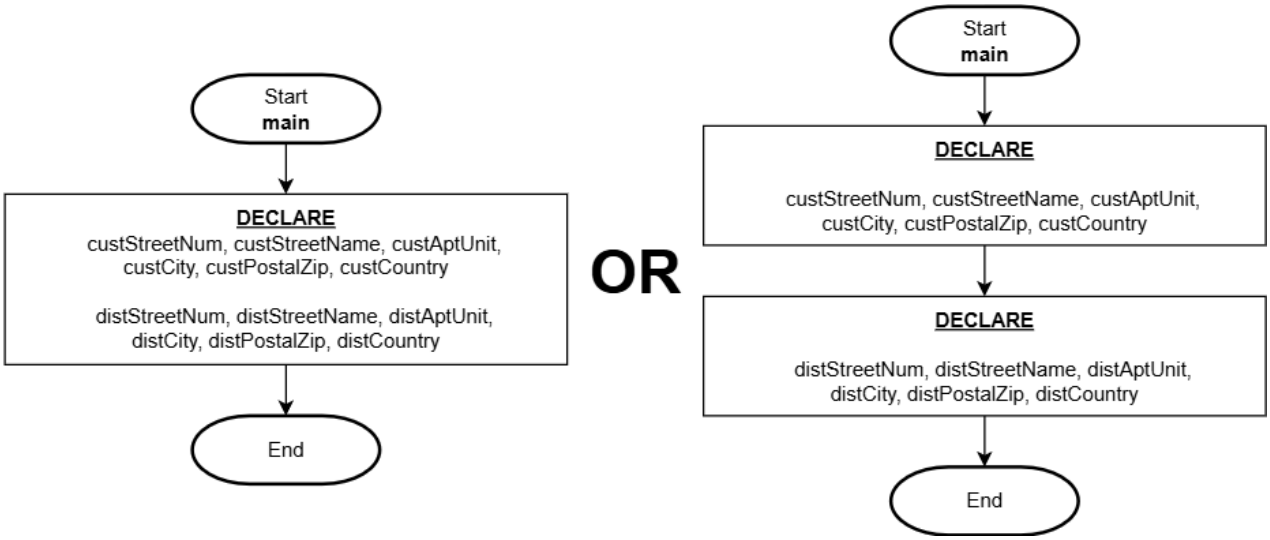
Sometimes we need to manage a lot of information which can lead to the use of many variables. It is a standard practice to **group logically related variables together** and as needed, using a combination of lines and line-spacing. Grouping variables help readers find variables quickly and this helps confirm you have variables for 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?

Pseudocode

```
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.

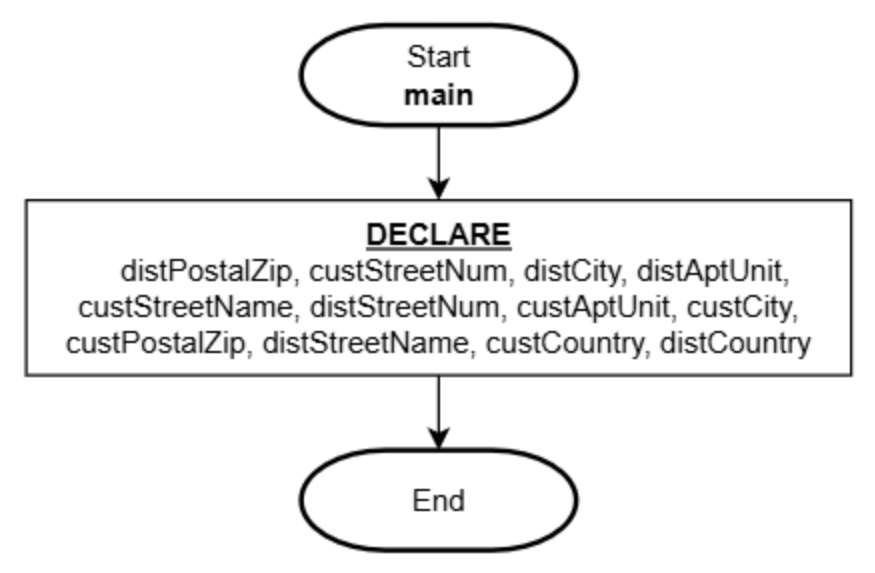
DON'T DO THIS

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:

Pseudocode

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

2. End
```

Flowchart


Start
main
DECLARE
distPostalZip, custStreetNum, distCity, distAptUnit,
custStreetName, distStreetNum, custAptUnit, custCity,
custPostalZip, distStreetName, custCountry, distCountry
End

Modularity with Functions

Overview

Modularity is an important aspect of programming as it provides flexibility in orchestrating logic needed for efficient reusability and efficient management of solution components. The most common component of modularity is the use of **functions**.

Functions represent algorithms comprised of several logical steps which perform a specific task. Functions are defined **once and in one place**. This is what makes it efficient and simpler to manage since modifications and trouble shooting bugs directs you to a single source.

When it's time to construct a complete solution, it is nice to be able to reference functions which do all the work (hiding the details) and simplifies the overall readability of the instructions so long as good naming practices have been applied.

Sometimes access to the composition of a function may not be possible. When you don't have access to the details of a function we refer to this as a **closed-box**.

> **REMINDER**
>
> As mentioned in the Computational Thinking sections on:
>
> - Decomposition: Functions are mostly identified when going through

the process of breaking things down into smaller logical parts.

- Pattern Recognition: Is another area where functions may be identified where you can extract a common piece of logic that can be reused in multiple places throughout the solution.
- Abstraction: Usually based on a pattern of functions where the concept is the same and can be combined to simplify the usability of a single function.

Function Identifiers (name)

Similarly, as variable identifiers should be well thought out and purposefully named to best describe the information it will be representing, so should the function naming process. **Variables are related to DATA**, but **functions are related to ACTION**. Therefore, **function identifiers should be well thought out and purposefully named to best describe their logical purpose and action**. Providing meaningful names to functions will contribute towards easier to read and manage logic (or code).

Reading logic that is applying functions should be like reading a standard sentence or close to it. There should be no mystery of WHAT the function's purpose is based on it's name however, like in variable naming practices, we need to be mindful of efficiency as we don't want to be referring to excessively long names (and in the case of programming code, we don't want to be typing them either!). That being said, unlike variable naming, function names can be a little longer given the importance of having the name best represent WHAT it does so we don't want to over abbreviate.

For example, if we need a function to write program data to a file in JSON format we could create a function with the following signature:

- Function: `WriteDataJSON (data)`
- Parameter: `data` is what will be written to file in JSON format
- Returns: TRUE (success) or FALSE (fail)

Here's an excerpt of how it would be called:

```
1. DECLARE:
      data
      result

2. -> 99. [ logic capturing a lot of information stored to 'data'
]

100. ASSIGN: result = WriteDataJSON (data)
101. Was the data successfully exported to a file?
      A. result=TRUE:
            1. DISPLAY:
                  "Data exported to JSON file!"
      B. result=FALSE:
            1. DISPLAY:
                  "FAILED to write to file."
102. End
```

Now let's consider a poorly named function that writes data to a JSON formatted file. If we replaced the name with `FormatDataJSON (data)`. While not totally incorrect, it does not reveal the fact that data is written to a file which the term "Write" successfully conveys. Another poor name would be `StoreData (data)` - this does not reveal the formatting standard used to save the data and the term **Store** can also refer to variables not files.

> **DON'T DO THIS**
>
> **Do NOT obfuscate** function names as this renders them meaningless to the logic if you can't easily understand a functions purpose based on the name alone:
>
> - a (data)
> - HorseGoat (data)
> - abcdef (data)
>
> These functions are not usable since we don't know what their purpose is based on their names!

Function Documentation

Functions should be **BRIEFLY** described to elaborate more on **WHAT** the purpose of the function is to expand on the name itself (the names alone are usually not enough). This would include **parameter(s)** (expected arguments) and **return value**.

This is a very important aspect of programming especially when you get into collaborative coding and sharing of custom libraries like API's (Application Programming Interface). Functions need to be summarized so the use of the function is clear and other programmers will know how to properly apply the function.

> **NOTE**
>
> For the purposes of these notes, **pseudocode functions** will be documented with a series of dashes: `------` before and after the

> description.
>
> See the below sub-section on Returning Information for some examples. Going forward, all examples will demonstrate function level documentation accordingly.
>
> **Flowchart functions** will almost always be the **main** function and generally will not require any documentation unless it is specific to an example requiring further detail.

Here is the template:

```
------------------------------------------------------------
Description/Purpose:
   [Provide a brief description of the function's purpose]

Argument(s) : [List each parameter/expected argument and if
               needed, a brief associated description]
              Example:
              [param-1: describe briefly what this is for]
              [param-2: describe briefly what this is for]

Return Value: [Describe possible return values if applicable]
------------------------------------------------------------

[FunctionName] ( [Parameter(s)] )

1. [list of steps...]
```

Closed-Boxes

There will be times when we need to use 3rd-party logic (known as **API's** :

Application Programming Interface) or other prepared logic from **system library functions** where we **don't have access to the details of how they work**.

These functions are known as "**closed-boxes**" (formerly "black boxes"). Closed-box functions are like "magic" because you call them and they do what you expect without explicitly knowing how it performed the task. It is important to note, sometimes this can be a problem because if there is a bug (error) or a missing piece of functionality in that piece of logic, there is nothing you can do to address it other than to redefine your own version of that logic in your own composed function.

> **HOW-TO**
>
> If you are calling a closed-box function from a flowchart, use the striped rectangle symbol to represent a complex function.

High-Level Functions

High-level functions are usually highly abstracted in that they can represent a lot of functionality/logic usually because this type of function will call many other functions to complete its task. This is like seeing the solution from "a birds eye view".

One such function that is commonly required in many programming languages is "**main**" which behaves as the entry-point to the application (or where the logic begins and typically ends).

> **NOTE**
>
> For the purposes of these notes, we'll be designating a `flowchart` **to**

> **always represent this higher-level view of the overall solution**. The directive of a flowchart is to describe an overall set of instructions in a simple to understand format for non-technical persons, so it is a natural and suitable application of a flowchart.

Low-Level Functions

Unlike high-level functions, lower-level functions are a lot more focused and detailed on a task that is highly limited in scope - most functions fit this category and are constructed to be reusable or to remove complexity from other larger scoped functions.

> ⚠ **NOTE**
>
> For the purposes of these notes, we'll be designating `pseudocode` **to always represent these lower-level detailed parts of the solution algorithm**.

Passing Information

Functions usually require information to be provided or sent to it to do its task. When a function requires information, it is constructed with one or more **parameters**. Parameter is a fancy term used that essentially means a variable.

For example, if we create a function that is responsible for displaying the date and time in a standard ISO 8601 format (YYYY-MM-DD HH:MM:SS), the function would require all six of those specific parts of data to be supplied:

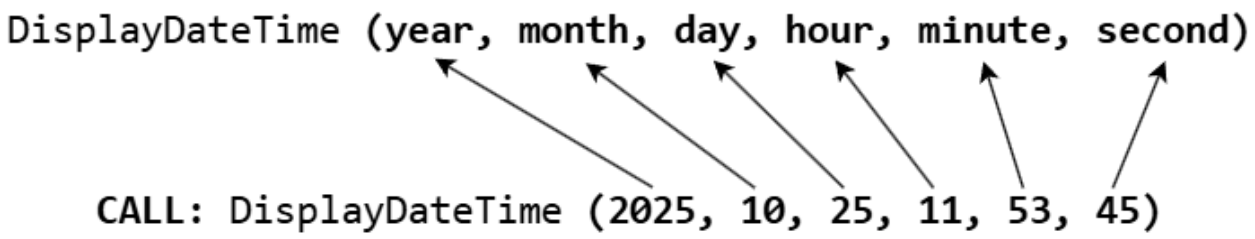
```
DisplayDateTime (year, month, day, hour, minute, second)
```

The comma delimited list of the date and time parts are the **parameters**. The parameters act as variables which can be used in the function logic to access the values sent to the function.

When it comes time to use this function, we will **CALL** it and supply the function with the information it requires (we call this "passing" data to a function) in the form of **arguments**. An argument is a value sent to a function.

Since the `DisplayDateTime` function has **six parameters**, we will need to send **six arguments** in the order it is expecting it:

```
CALL: DisplayDateTime (2025, 10, 25, 11, 53, 45)
```



Each argument sent to the function will be assigned to the corresponding parameter variable and then the function would construct the output assembling the variables into their appropriate sequence:

```
year-month-day hour:minute:second
```

The expected outcome of this function call, would be to display the date time data as:

```
2025-10-25 11:53:45
```

Returning Information

Functions often do more than just an explicit task - it is quite common for information to be returned to the caller of the function (ex: back to where the function was called from). Most programming languages support flexible methods in how information can be returned from a function:

1. Explicit
2. Implicit
3. Both explicit and implicit

Let's have a look at these methods and how they will be documented in pseudocode.

> **REMINDER**
>
> Since these notes will be limiting the use of flowcharts to the `main` overview process only, we won't need to concern ourselves with documenting return values for flowcharting.

Explicit Return

Returning information from a function **explicitly** is accomplished by using the keyword: **`return`**. This method provides a **single variable of information** to be returned from the function.

For example, if we have a function: **`Multiply (number, multiple)`** which will return the results of multplying the provided `number` by the provided

`multiple`, the logic would look like the following:

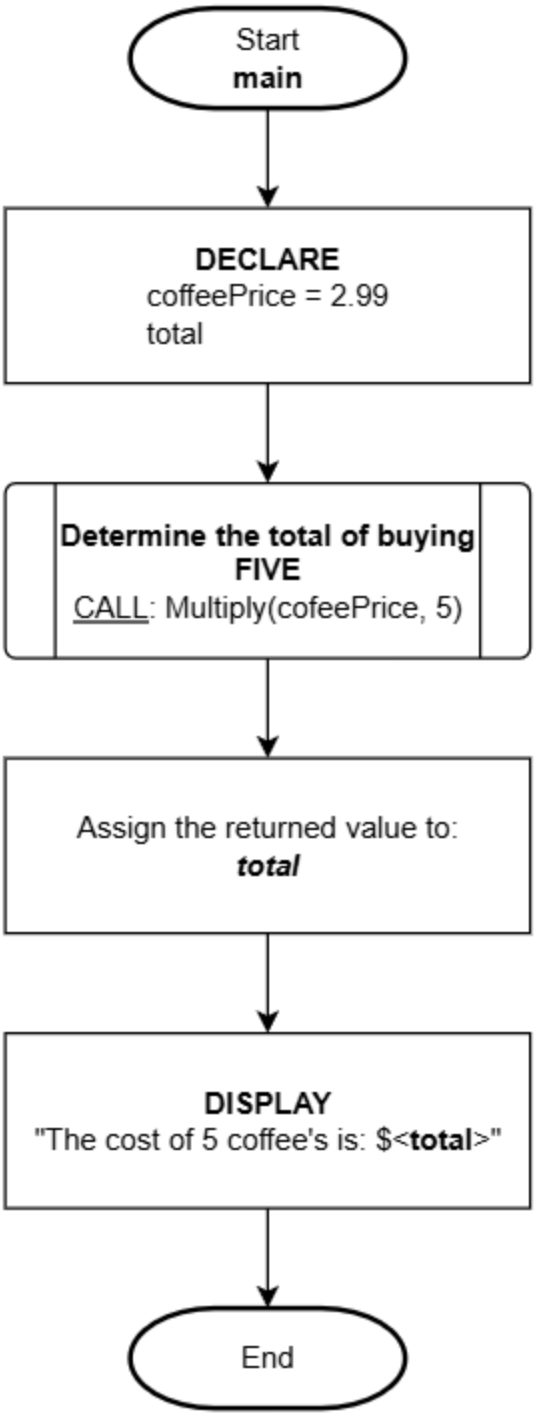
Function: `Multiply (number, multiple)`

```
------------------------------------------------------------------
Description/Purpose:
  Demonstration of a function that returns a value implicitly.

Argument(s) : number (operand 1 to be multiplied)
              multiple (operand 2 to be multiplied)
Return Value: result of the multiplication
------------------------------------------------------------------

Multiply (number, multiple)

1. DECLARE:
      result

2. ASSIGN: result = number * multiple

3. RETURN result

4. End
```

This can be refined by not creating a variable:

```
Multiply (number, multiple)

1. RETURN number * multiple

2. End
```

Here is how the function can be used from a **flowchart**:



Here is how the function can be used from **pseudocode**.

Function: `main()`

```
-----------------------------------------------------------------
Description/Purpose:
  Demonstration of calling a function "Multiply" that returns
  a value explicitly.

Argument(s) : NONE
Return Value: NONE
-----------------------------------------------------------------

main()

1. DECLARE:
      coffeePrice = 2.99
      total

2. ASSIGN: total = CALL: Multiply(coffeePrice, 5)

3. DISPLAY:
      "The cost of 5 coffee's is: $<total>"

4. End
```

Implicit Return

Returning information implicitly involves updating a parameter variable. The understanding with this method is that any changes made to a parameter variable will directly affect the original. This can simplify how return values are documented but also can lead to missed data changes since it is not as clear.

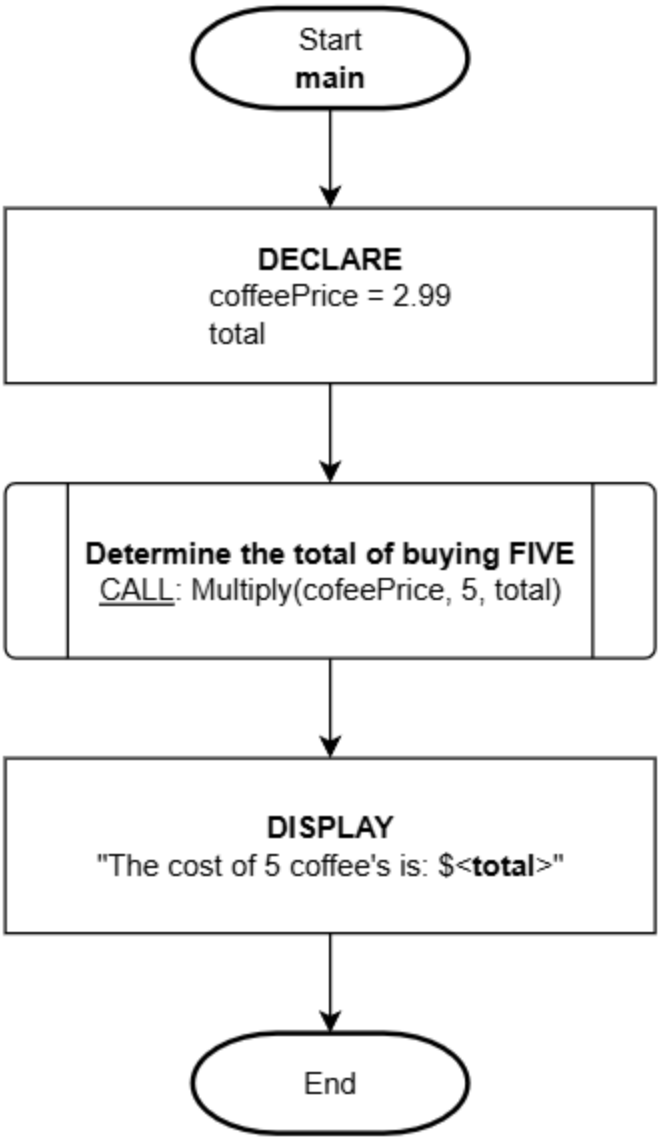
Let's do another version of the preceding example using this method (**notice the extra parameter `total`**).

Function: `Multiply (number, multiple, total)`

```
------------------------------------------------------------------
Description/Purpose:
  Demonstration of a function that returns a value implicitly.

Argument(s) : number (operand 1 to be multiplied)
              multiple (operand 2 to be multiplied)
              total (result of multiplication)
Return Value: NONE
------------------------------------------------------------------

Function: Multiply (number, multiple, total)

1. ASSIGN: total = number * multiple

2. End
```

The `total` parameter variable was used as the return value from the function by assigning the result of the calculation to that variable.

Here is how the function can be used from a **flowchart**:



Here is how the function can be used from **pseudocode**.

Function: `main()`

```
------------------------------------------------------------------
Description/Purpose:
  Demonstration of calling a function "Multiply" that returns
  a value implicitly.
```

The advantage of implicit returns is it is not limited to a single variable return value like the explicit method. The **implicit method, can return multiple values through more than one parameter variable**!

> **NOTE**
>
> Some programming languages refer to implicit return values through parameters as:
>
> - output parameters
> - pointers
> - references

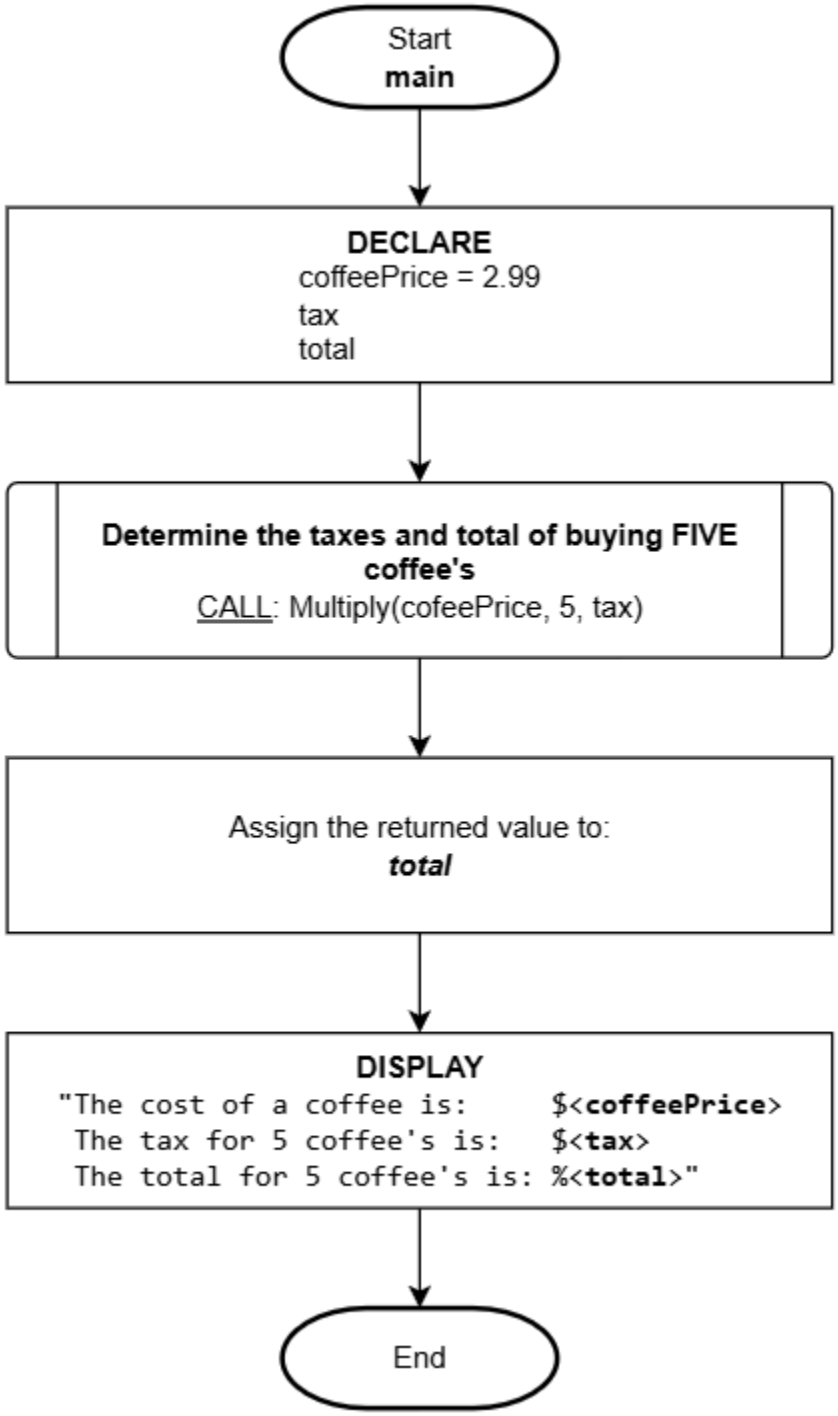
Both Explicit AND Implicit Returns

A combination of explicit and implicit methods can be done too! Expanding on the preceding examples, we will add a tax component. The calculated tax amount will be returned implicitly via the tax parameter and the total (including tax) will be returned explicitly:

Function: `Multiply (number, multiple, tax)`

```
------------------------------------------------------------------
Description/Purpose:
  Demonstration of a function that returns values applying both
  the implicit and explicit methods.

Argument(s) : number (operand 1 to be multiplied)
              multiple (operand 2 to be multiplied)
              tax (result of tax portion)
Return Value: total including tax
------------------------------------------------------------------
```

Here is how the function can be used from a **flowchart**:



Here is how the function can be used from **pseudocode**:

Function: `main()`

```
------------------------------------------------------------------
Description/Purpose:
  Demonstration of calling a function "Multiply" that returns
  values applying both implicit and explicit methods.

Argument(s) : NONE
Return Value: NONE
------------------------------------------------------------------

main()

1. DECLARE:
      coffeePrice = 2.99
      tax
      total

2. ASSIGN: total = CALL: Multiply(coffeePrice, 5, tax)

3. DISPLAY:
      "The cost of a coffee is:     $<coffeePrice>
       The tax for 5 coffee's is:   $<tax>
       The total for 5 coffee's is: $<total>"

4. End
```

WARNING

As you can see in this example, it can be easy to "miss" how the `tax` is assigned because it is IMPLICITLY updated by the function.

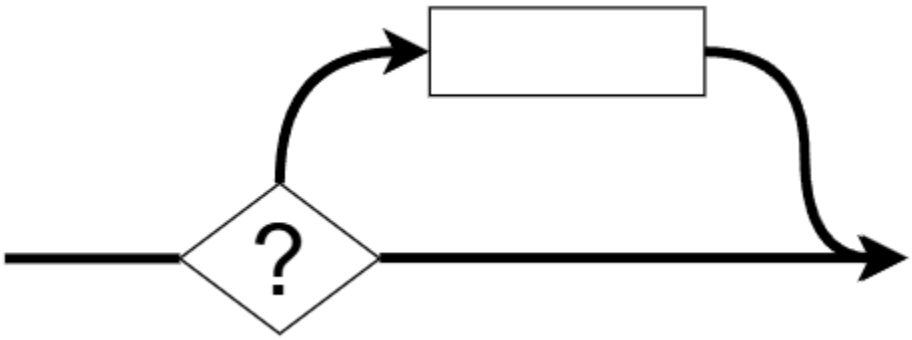
Logic - Selection

Overview

Computers programs are not very useful without the ability to adjust to changing conditions. A mechanism is therefore needed to enable programs the ability to evaluate when to change logical direction which can include executing specific logic and/or to avoid specific logic.

The concept of **Selection** is what provides programs the ability to apply logical decisions which can alter the execution sequence. There are several variations of selection (especially when coding it in a specific computer language), however the main three logical flows will be examined here.

Optional Selection



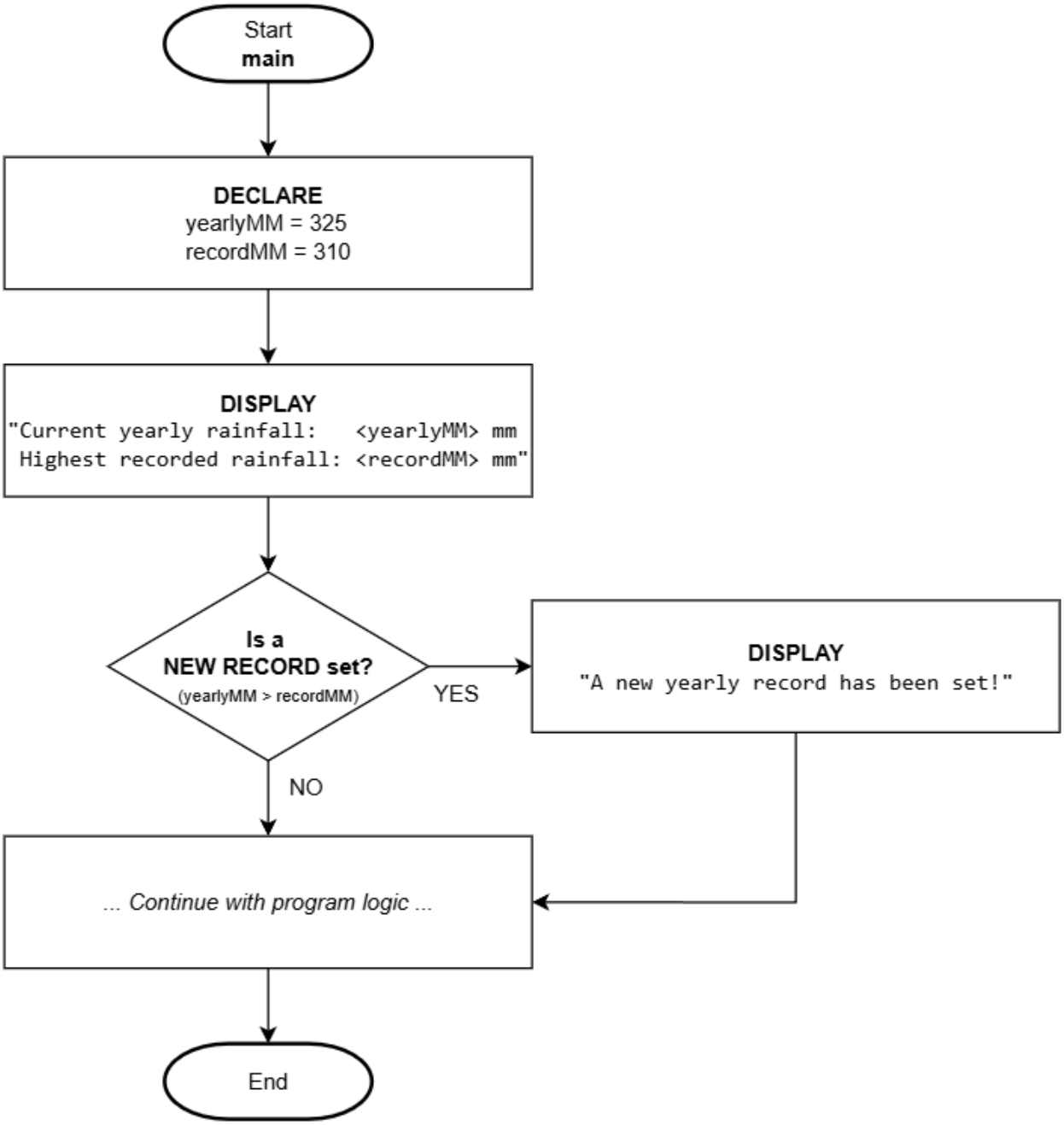
Optional selection provides a program the ability to execute **additional logic** to do something **only if one or more conditions are determined TRUE**, otherwise, the main program logic continues as normal.

Let's review part of a program where the total yearly rainfall is being evaluated to see if it has set a new record. The program should update the highest record

only when the yearly rainfall amount is determined to be more than the last known record. The logic to update the record should only occur if the yearly rainfall has been determined to be greater than the last known record. If the yearly rainfall is 325 mm and the last known record is 310 mm, we should expect the logic to update the record to the new value, otherwise, do nothing (continue with the normal program logic).

Flowchart

Here is how it would be done in a flowchart:



NOTE

- Notice the **DIAMOND shape** for the **selection decision**.
- The question is using **concise non-technical language**.

- The technical part (optional) has less emphasis and is in smaller font.
- Something "extra" only occurs if it is evaluated to TRUE (yes).

Pseudocode

Here is the pseudocode equivalent.

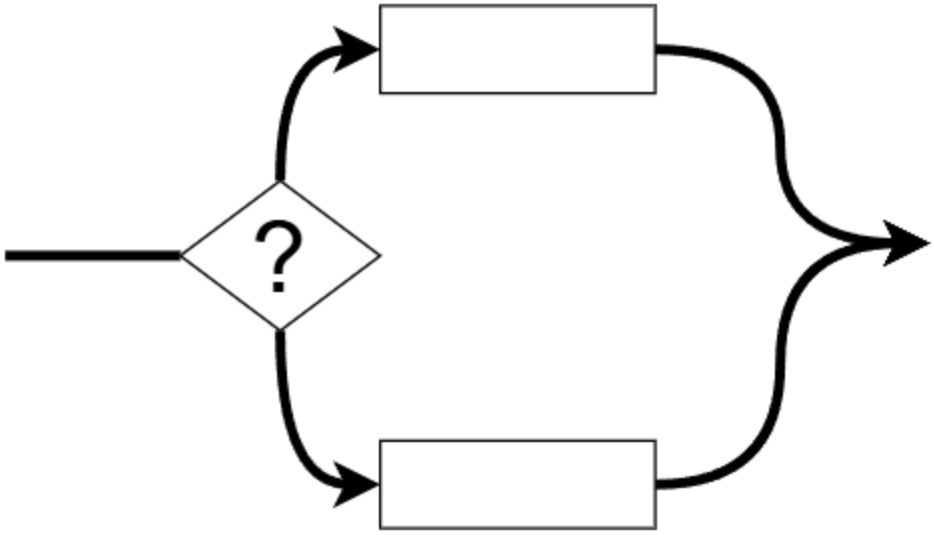
Function: `main()`

```
------------------------------------------------------------------
Description/Purpose:
Demonstration of optional selection.

Argument(s) : NONE
Return Value: NONE
------------------------------------------------------------------

main()

1. DECLARE
      yearlyMM = 325
      recordMM = 310

2. DISPLAY:
      "Current yearly rainfall:   <yearlyMM> mm
       Highest recorded rainfall: <recordMM> mm"

3. Is yearlyMM > recordMM ?
      A. YES:
            1. ASSIGN: recordMM = yearlyMM
            2. DISPLAY:
                  "A new yearly record has been set!"
```

The program will **only (optionally)** update the record and display a message indicating a new record was set **if the current year's rainfall is higher** than the previous set record. The expression in **step #3 is "selection"** where a `TRUE` or `FALSE` condition is being evaluated and based on that determination, extra logic may be executed.

> ⚠ **NOTE**
>
> - Notice the text 'label' `YES:` used to identify the answer to the selection question?
> - Notice the logic indentation (`TAB`) consistency of the pseudocode?
> - Notice how the enumeration changed from numbers (1...2...) to characters (a...b...) when the 'inner logic' is sequenced?
>
> It is important to maintain easy to read logic in pseudocode especially when **nested statements are involved**.
>
> The nested statements in this example is where the group of statements are placed under the `YES:` logic path.

Alternative Selection

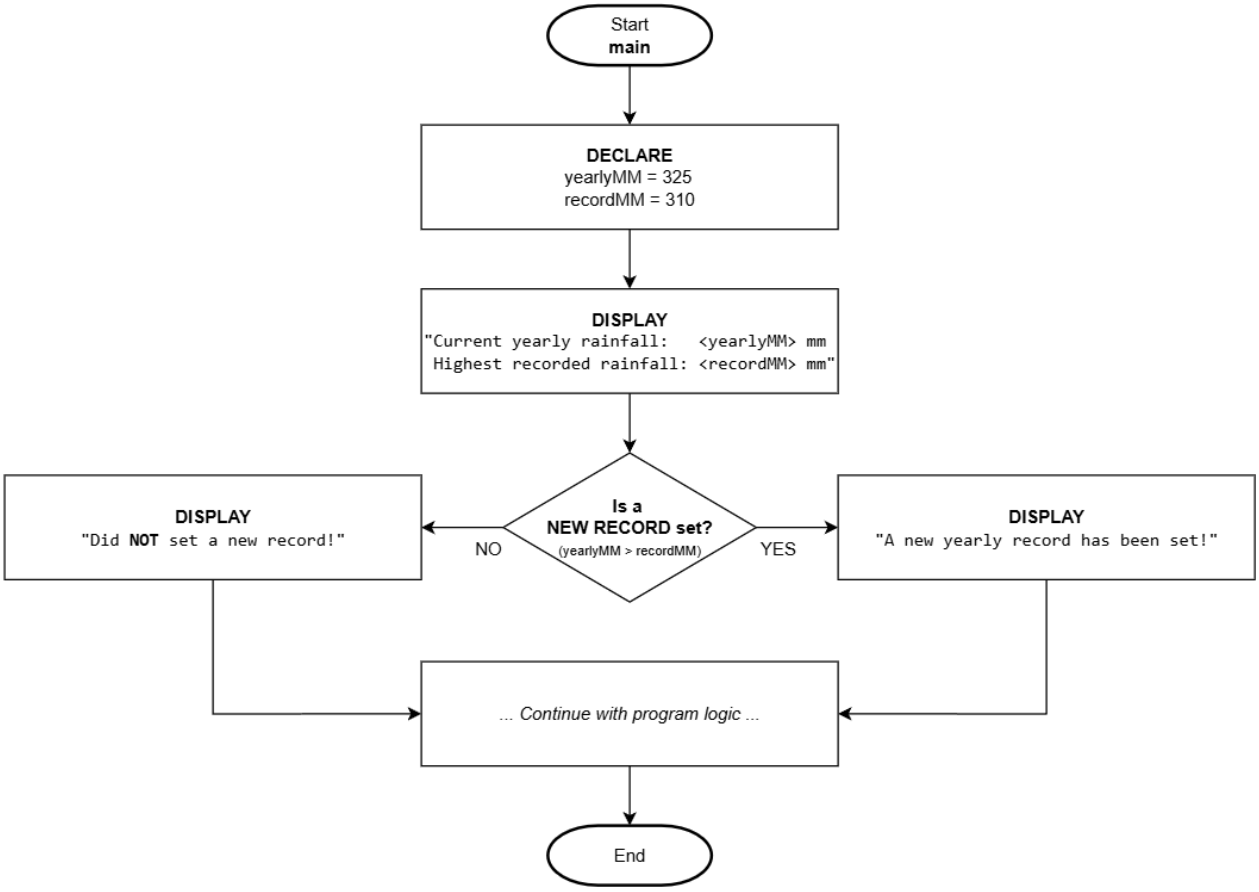


Alternative selection is like a 'Y' in the road where **a decision must be made** and no matter what **only one of two different logic paths** will be executed but **not both**.

Continuing with the preceding example, the application will now display an outcome no matter what. **EITHER** a new rainfall record was set, **OR** a new record was NOT set.

Flowchart

Here is how it would be done in a flowchart:



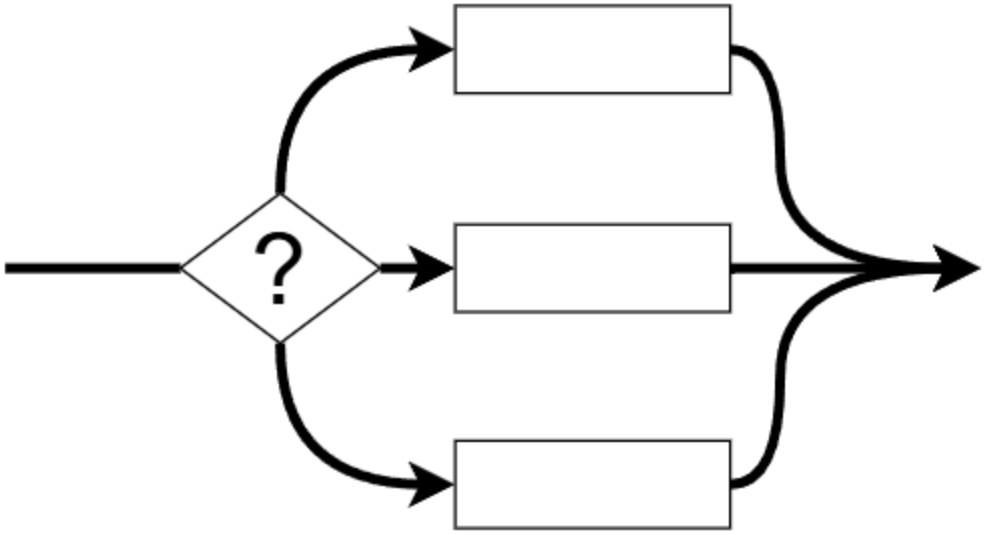
Pseudocode

And here is how it would be done in pseudocode.

Function: `main()`

```
------------------------------------------------------------------
Description/Purpose:
Demonstration of alternative selection.

Argument(s) : NONE
Return Value: NONE
------------------------------------------------------------------
```

Multiple Alternative Selection



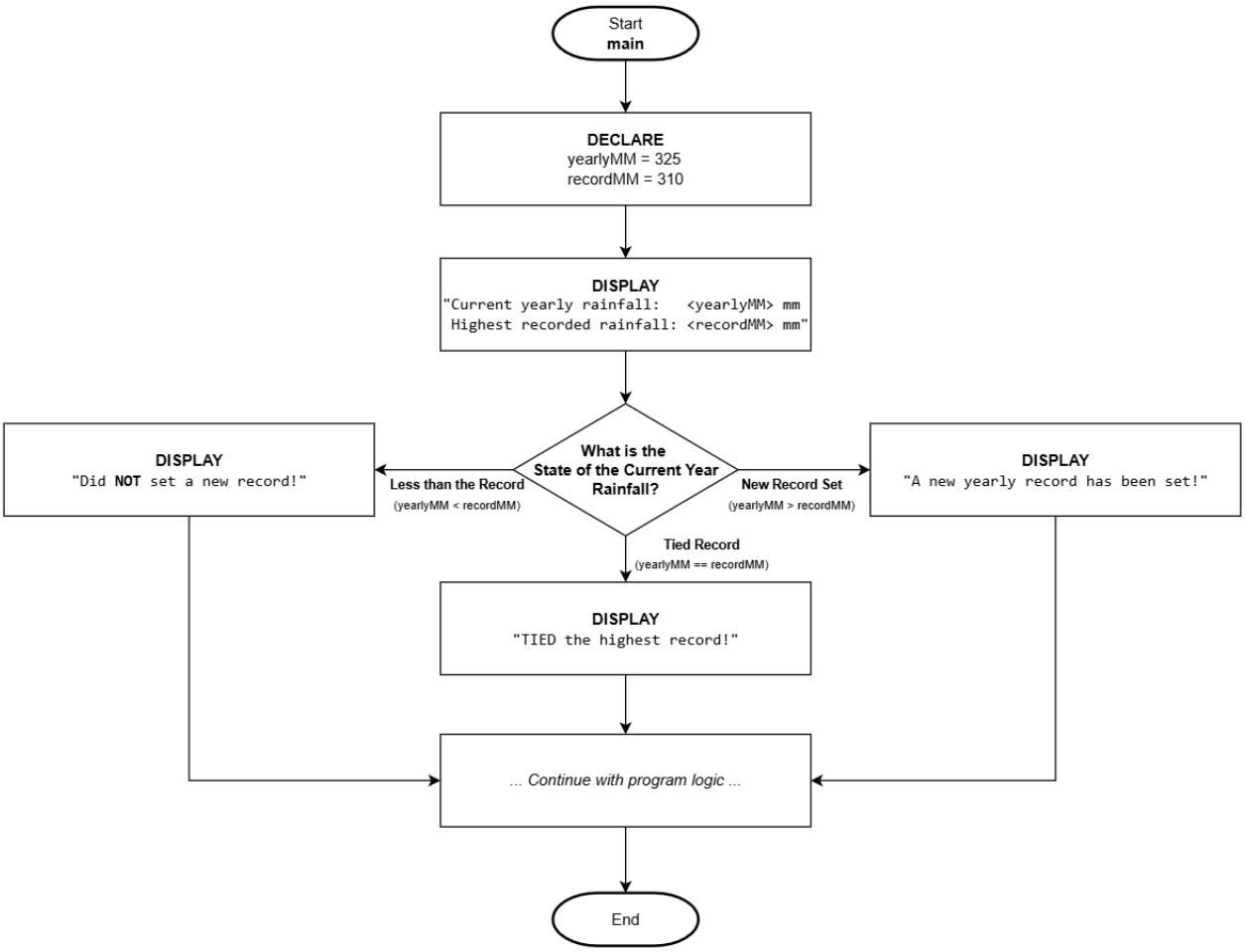
Multiple alternative selection is like an 'E' in the road where **a decision must be made** and no matter what **only one** of **THREE or more** different logic paths will be executed but **not all**.

Continuing with the preceding example, the application will display an outcome no matter what for **only one** of these **three possibilities**:

1. A **new rainfall record** was set.
2. The current year rainfall **TIED** with the highest record.
3. A new record was **NOT set**.

Flowchart

Here is how it would be done in a flowchart:



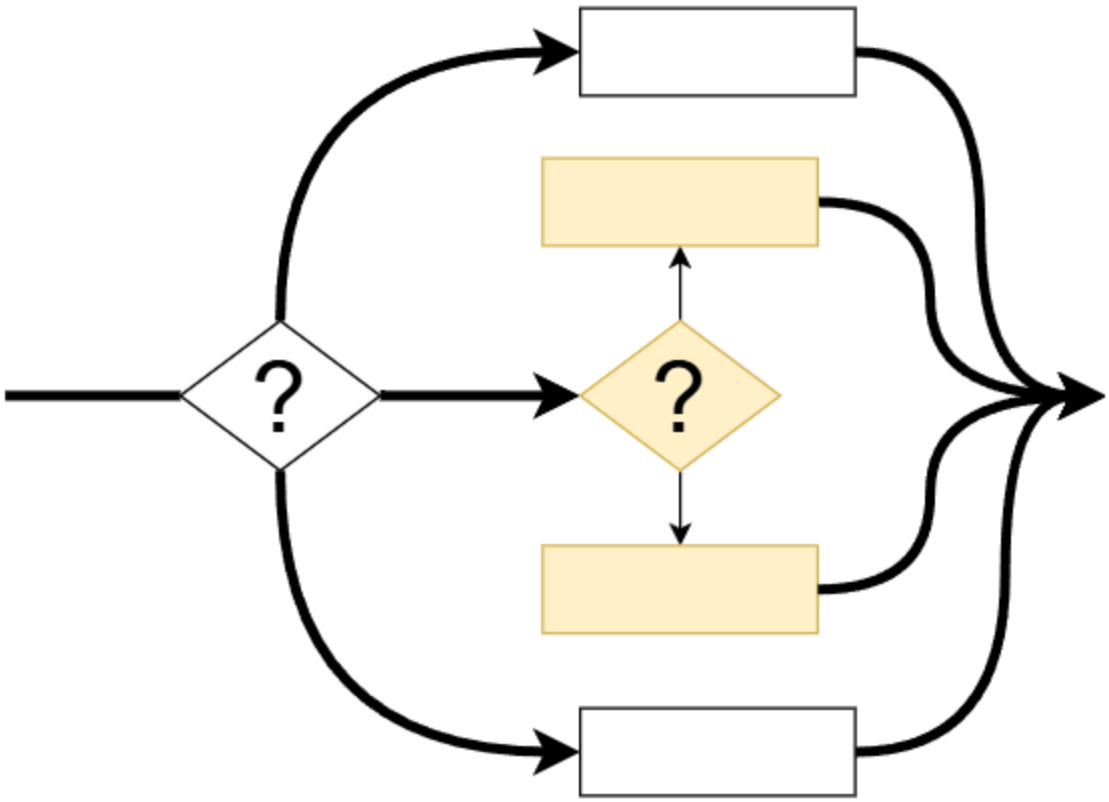
Pseudocode

And here is how it would be done in pseudocode.

Function: `main()`

```
------------------------------------------------------------------
Description/Purpose:
Demonstration of multiple alternative selection.

Argument(s) : NONE
Return Value: NONE
```

Nested Selection



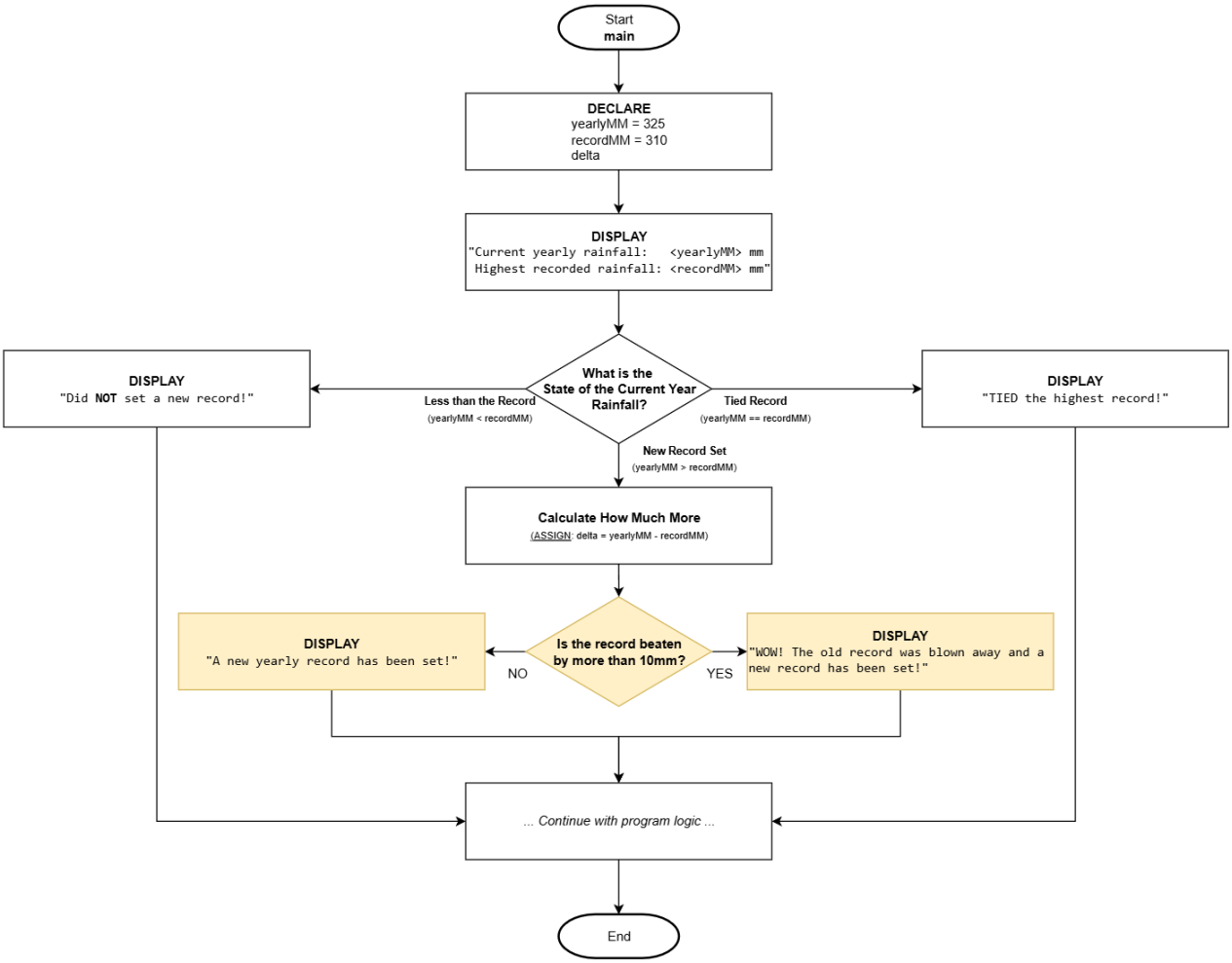
Nesting is a way to sequence statements or logic constructs such as selection and iteration **within another logical construct**. It is a way to group statements or other logic under a dependency. In the case of selection, this would mean anything intended to execute when the selection statement is evaluated to TRUE, would be nested within the limits of the selection.

Using the preceding example on yearly rainfall, let's say we want to have different levels of "excitement" when a new rainfall record is determined. If the old record is beat within 10 mm (inclusive), then we want to state: "A new record has been set!", but when it is beaten by more than 10 mm, we want it to state: "WOW! The old record was blown away and a new record has been

set!".

Implementing this, would require logic to be placed within the section where it is determined the record was beaten - this would be nested selection!

Flowchart



The golden highlighted section encompasses the **nested selection** which is only executed if the parent selection that tests for a beaten record evaluates to TRUE.

Pseudocode

Function: `main()`

```
------------------------------------------------------------------
Description/Purpose:
Demonstration of nested selection.

Argument(s) : NONE
Return Value: NONE
------------------------------------------------------------------

main()

1. DECLARE
      yearlyMM = 325
      recordMM = 310
      delta

2. DISPLAY:
      "Current yearly rainfall:   <yearlyMM> mm
       Highest recorded rainfall: <recordMM> mm"

3. What is the state of the current year rainfall?
      A. NEW RECORD SET (yearlyMM > recordMM):
            1. ASSIGN: delta = yearlyMM - recordMM
            2. ASSIGN: recordMM = yearlyMM
            3. Is the record beaten by more than 10mm?
                  A. YES (delta > 10):
                        1. DISPLAY:
                              "WOW! The old record was blown away
and a new record has been set!"
                  B. NO:
                        1. DISPLAY:
                              "A new yearly record has been set!"
```

Step `3.` is the **outer selection** (parent) and sub-step `3.A.3.` within the "NEW RECORD SET" section is the **nested selection**. Nesting is only executed if the parent (outer) dependency is evaluated to TRUE.

Testing

As referred to in the computational thinking - testing section, "what-if" scenarios are used to help test the logic and expected outcomes. **Selection is pivotal to controlling program execution** to take the necessary logic paths required to address all reasonable possibilities and certainly the expected paths to address the problem.

Semantic failures are often directly tied to selection logic, so it is important you build strong selection skills - these skills will also help you be a more efficient and effective problem solver and programmer.

Iteration

Overview

It is very common to have to repeat logic, but as we know from the computational thinking model (pattern recognition) approach, placing detailed logic into a function when we know it will be reused or repeated is a good practice, but is it enough?

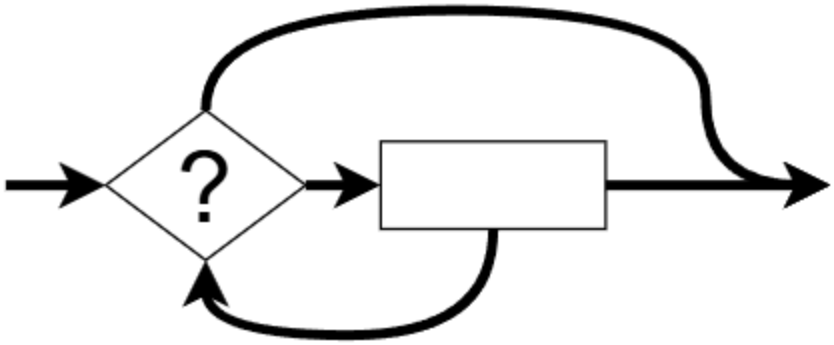
Let's say we need logic to manage a list of items for a shopping list. We would need repeating logic responsible for getting the user input detail for an item and then add each item to the shopping list (this would be a good use of a function), but how many items will there be? Will a shopping list always have the same number of items? Probably not! We can't limit the logic to sequence a set number of calls to the function that adds a list item because that would be a pretty useless application (what if you want fewer or more?)!

The problem is **how to structure the repeating logic** so it can be executed **any number of times (if at all)** and yet **refer to the logic only ONCE**. This is where the **iteration (looping) construct** comes in!

To manage a varying possible number of repetitions (iterations/loops), we must apply **iteration constructs**. There are many forms of these constructs in specific programming languages, but the abstraction (idea/concept) itself is the same with only minor semantic differences. These notes will be focused on the pure concept of iterating (looping) covering two forms:

1. Optional Iteration
2. Mandatory Iteration

Optional Iteration



Optional iteration is the most common. This involves **checking a condition BEFORE** the possible repeating logic to determine if looping is even required and if so, iterate **until no more iterations are required**. This could result in:

- No iterations
- One iteration
- Many iterations
- Possible infinite iterations

> **TECHNICAL INSIGHT**
>
> **DISCLAIMER**: Something to note about iteration constructs, is that all forms can be made to work like the others however each one has its designed intentions.
>
> Optional iteration found in most programming languages include:
>
> - `while`
> - `for`

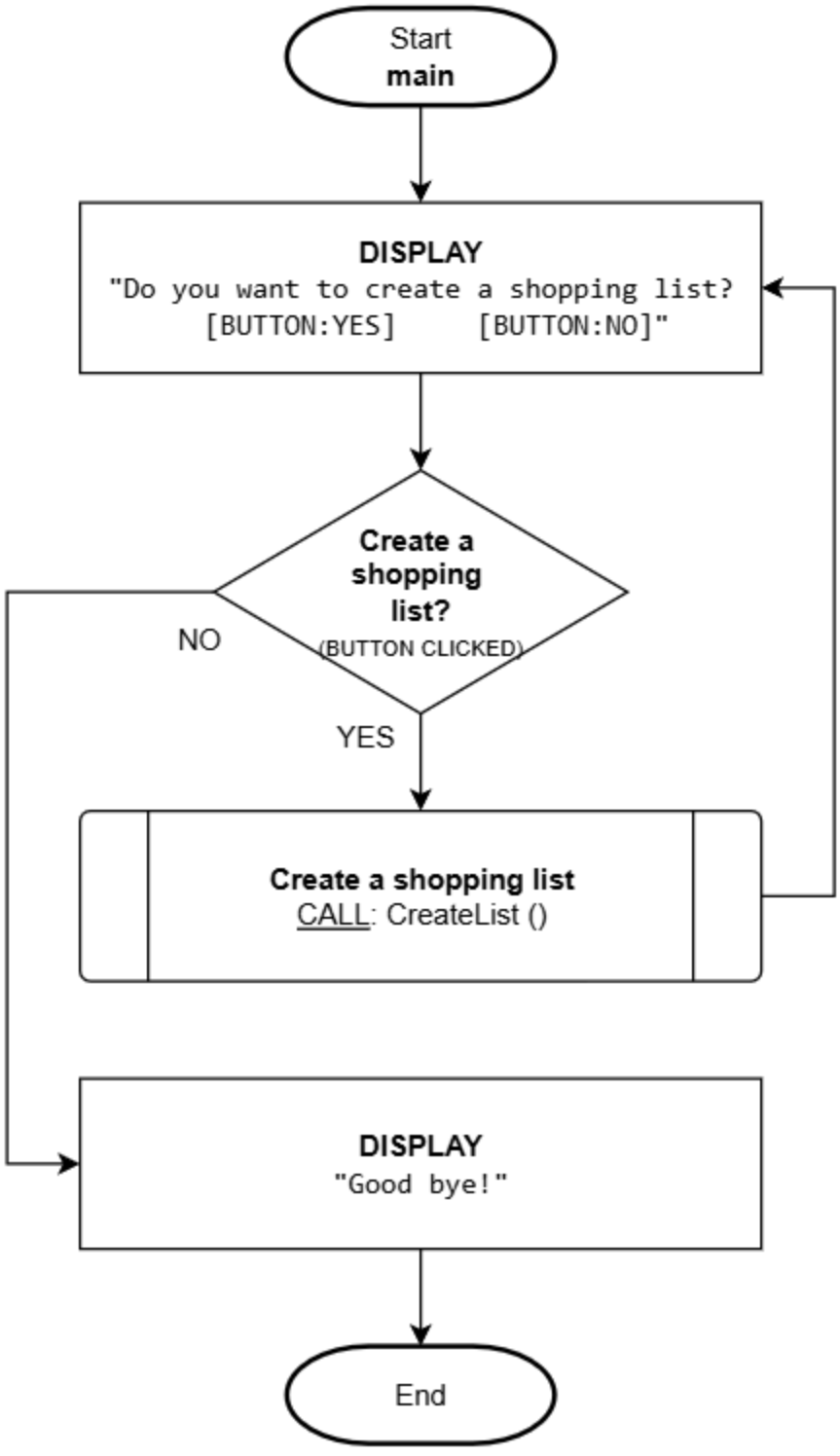
- `foreach`

NOTE: These are language-specific terms and will not be used in the documentation of logic in these notes.

Flowchart

Let's continue with the shopping list theme from the introduction. We want to create an application that will allow a user to create as many shopping lists as they want - if any. This will involve iteration since the user can make many lists but we don't know how many. **Because this logic approach is optional-based, this means it is possible a shopping list may never be created or many may be created**.

> **NOTE**: To help with this example, we will be using a **closed-box function** to deal with the details of creating a shopping list where items are added to a list. Here's an overview of the function:
>
> - **Name**: `CreateList ()`
> - This function prompts the user to enter as many items as they wish to a shopping list.
> - The function logic details will be described in the pseudocode section example.


Start
main
DISPLAY
"Do you want to create a shopping list?
[BUTTON:YES] [BUTTON:NO]"
Create a shopping list?
(BUTTON CLICKED)
NO
YES
Create a shopping list
CALL: CreateList ()
DISPLAY
"Good bye!"
End

Pseudocode

> **NOTE**: We will be using a **closed-box function** that will deal with the details of obtaining item information and adding it to a list of items. Here's an overview of the function:
>
> - **Name and parameter**: `AddItem (itemList)`
> - **itemList**: Parameter represents the entire shopping list and will implicitly add an item to that list

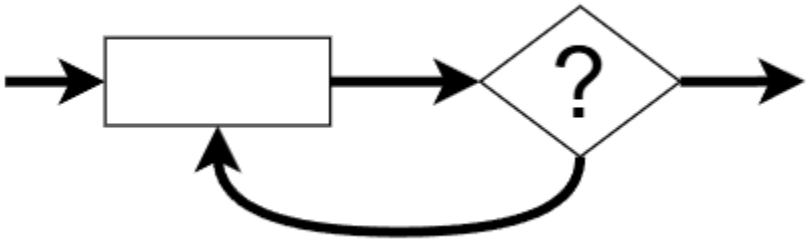
Function: `CreateList()`

```
------------------------------------------------------------------
Description/Purpose:
  Demonstration of optional iteration where zero or more shopping
  items can be added to the list.

Argument(s) : NONE
Return Value: NONE
------------------------------------------------------------------

CreateList()

1. DECLARE:
      shoppingList
      itemCount = 0

2. DISPLAY:
      "Do you want to add an item to the shopping list?
          [BUTTON: YES]     [BUTTON: NO]"

3. Keep adding items to the shopping list - which button was
selected?
```

- The repeating construct begins in **step #3**. This iteration may never be executed because it is dependent on the preceding answer to the question of adding an item to the shopping list. If the user selects the NO button the first time, then no items will be added to the list however, as long as the user selects the YES button it will continue iterating adding more items to the shopping list. As soon as the user selects the NO button, the iteration will stop.

- **Step 3.A.3.** is critical as it directs the next executed piece of logic back to the add another item prompt where repetition will repeat from.

Mandatory Iteration



Mandatory iteration is almost the same as optional iteration with the only difference being in the placement of the condition. Unlike optional iteration where the condition is checked first, in mandatory iteration **the condition is placed AFTER** the logic that would be repeated and will iterate **until no more iterations are required**. This results in:

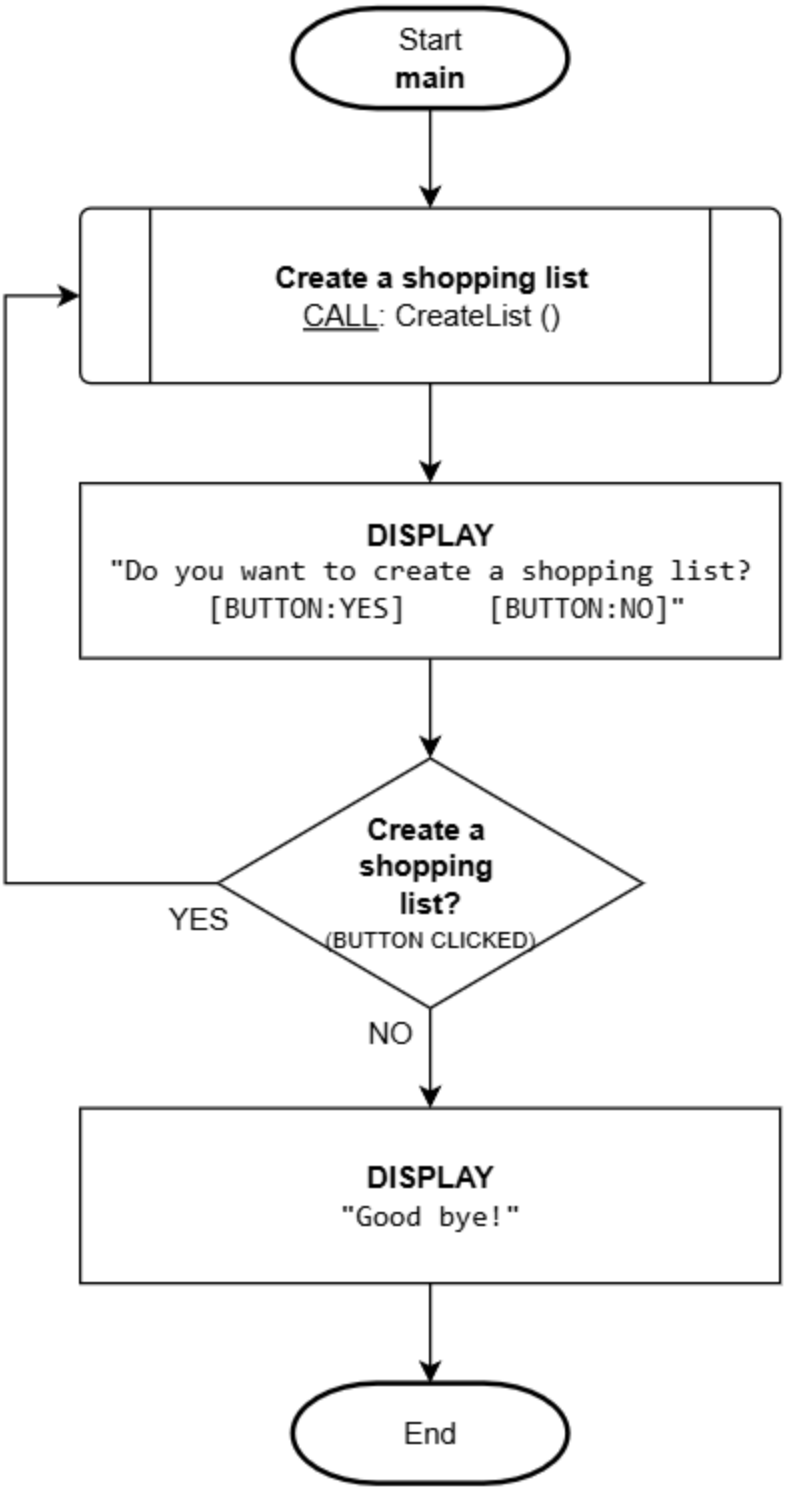
- **At least ONE iteration**
- Many iterations
- Possible infinite iterations

Mandatory iteration is usually applied in `validation routines` where actions must occur at least once before determining if repeating is required. This form involves executing some logic first, then **checking a condition AFTER** to determine if looping should be repeated **until no more iterations are required**.

> **TECHNICAL INSIGHT**
>
> **DISCLAIMER**: Something to note about iteration constructs, is that all forms can be made to work like the others however each one has its designed intentions.
>
> Mandatory iteration found in most programming languages include:
>
> - `do ... while`
>
> **NOTE**: This is a language-specific term and will not be used in the documentation of logic in these notes.

Flowchart

Let's modify the preceding OPTIONAL iteration flowchart example to be **MANDATORY iteration**. This change will **mandate the user to enter at least one shopping list** (unlike before where none was possible).



> Notice the user is forced to create at least one shopping list because the option of creating a shopping list was moved AFTER the logic that creates

> a list.

Pseudocode

Let's modify the preceding OPTIONAL iteration pseudocode example to be **MANDATORY iteration**. This change will **mandate the user to add at least one item to the shopping list** (unlike before where none was possible).

Function: `CreateList()`

```
-----------------------------------------------------------------
Description/Purpose:
  Demonstration of mandatory iteration to add at least one
  shopping item to the list.

Argument(s) : NONE
Return Value: NONE
-----------------------------------------------------------------

CreateList()

1. DECLARE:
      shoppingList
      itemCount = 0

2. CALL: AddItem (shoppingList)
3. ASSIGN: Add 1 to itemCount

4. DISPLAY:
      "Do you want to add another item to the shopping list?
          [BUTTON: YES]     [BUTTON: NO]"

5. Keep adding items to the shopping list - which button was
selected?
      A. YES:
```

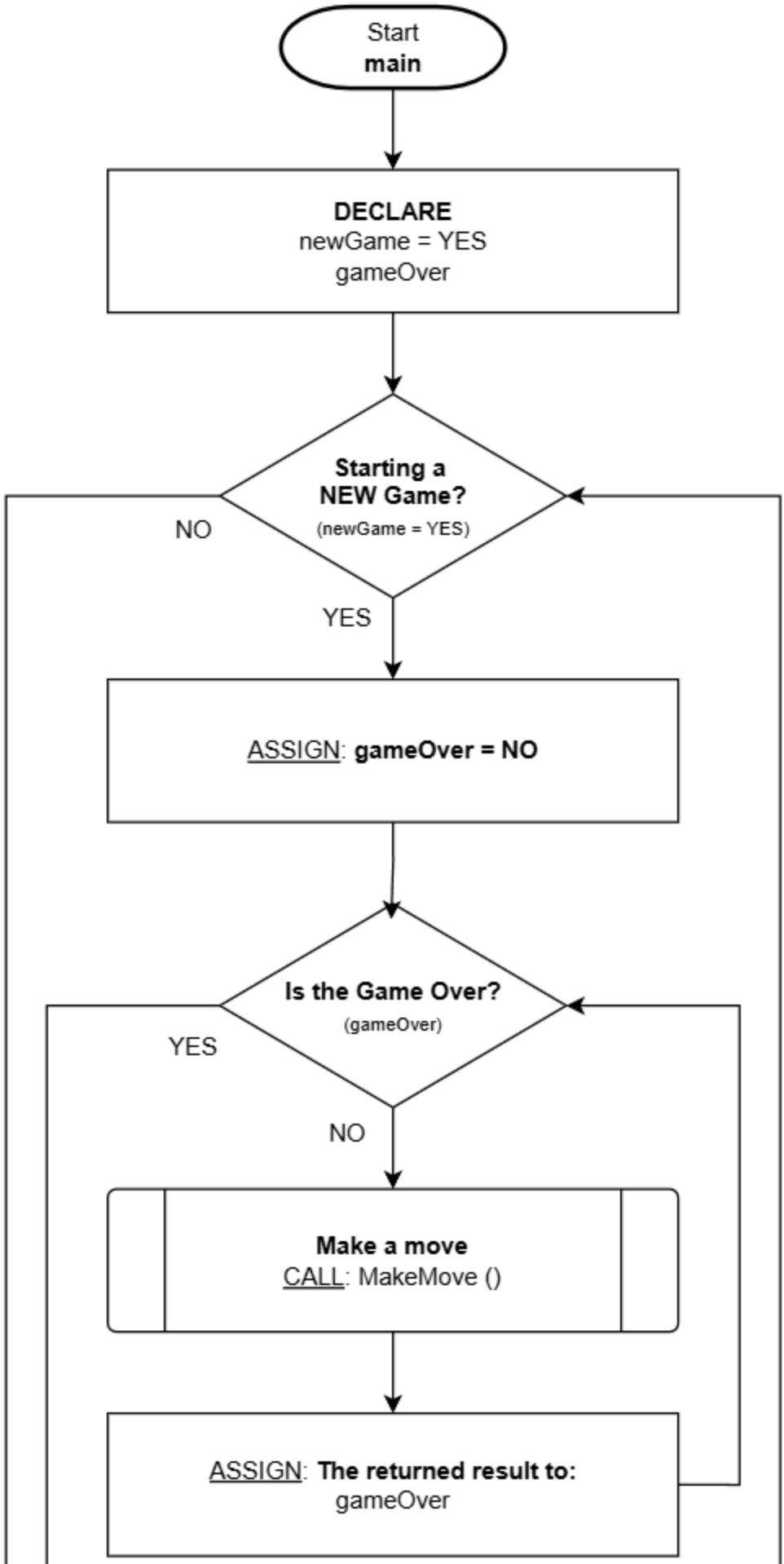
> No matter what, we want the user to **add at least one item to the shopping list** so we start with the logic that implements that. We don't ask the user for the option to add another item until **AFTER one has been added** and from there, the user can continue to add as many items as they wish (repeating from step #2).

Nested Iteration

Nested iterations follow the same concept as a nested selection from the previous topic - it is where an **iteration logic construct is placed within another construct**.

The below example shows a main iteration for a game (outer/parent loop) and within a game, there is a loop of player moves (this is the **nested iteration**). The outcome of a "MakeMove" function call determines if the game is over. When the game is over, a new game can be started and the entire process repeats.

Flowchart


Start
main
DECLARE
newGame = YES
gameOver
Starting a
NEW Game?
(newGame = YES)
NO
YES
ASSIGN: gameOver = NO
Is the Game Over?
(gameOver)
YES
NO
Make a move
CALL: MakeMove ()
ASSIGN: The returned result to:
gameOver

Pseudocode

Here is the pseudocode equivalent.

Function: `main()`

```
------------------------------------------------------------------
Description/Purpose:
Demonstration of nested iteration.

Argument(s) : NONE
Return Value: NONE
------------------------------------------------------------------

main()

1. DECLARE
      newGame = YES
      gameOver

2. Start a new game?
      A. YES (newGame=YES):
            1. ASSIGN: gameOver=FALSE
            2. Game over?
                  A. NO (gameOver=FALSE):
                        1. ASSIGN: gameOver = CALL MakeMove()
                        2. REPEAT: from Step #2.A.2

            3. DISPLAY:
                  "Start a new game?
                  [BUTTON:YES]   [BUTTON:NO]"

            4. ASSIGN: newGame = Selected BUTTON
            5. REPEAT: from Step #2

3. DISPLAY:
      "Thanks for playing! Bye!"
```

Key Parts:

Step `2.` is the main game loop (outer/parent)

Step `2.A.2.` is the **NESTED** player moves loop that occurs within a game until it's over

Data Containers & Collections

Overview

The simplest form of a data **container** is an array. The concept of an array is a variable capable of storing many values. All programming languages support this simplest form of a container, but many object-oriented languages such as C++, C#, Java, Python, etc., have additional variations mostly referred to as **collections** which provide an additional layer of operations (functions) to help simplify the navigation and management of the data. For the sake of these notes, and to maintain the language-agnostic theme, we will merge the concepts of these two major types of data representation as one and refer to it as a **collection** for simplification and consistency.

Collection Functions

To simplify how to use the functionality of a collection, this section will define a limited list of what features a collection provides so we can refer to this as needed in all future examples.

Navigation Actions

Action	Explanation
FIRST	This returns the **first** item in the collection and updates the

Action	Explanation
	current position. If there are no items in the collection, it returns an **EMPTY** item.
LAST	This returns the **last** item in the collection and updates the current position. If there are no items in the collection, it returns an **EMPTY** item.
NEXT	This returns the **next** item in the collection and updates the current position. If no more items remain, it returns an **EMPTY** item.
PREV	This returns the **previous** item in the collection and updates the current position. If the current item is the first item, it will return an **EMPTY** item.
AT	This returns the item **at the position stated** (zero-based) in the collection and updates the current position. This requires an **argument** for the position and if there is no item, it will return an **EMPTY** item.
COUNT	This returns the number of items in the collection.

NOTE: The default current position is assumed to be at the beginning of the list.

Manipulation Actions

Action	Explanation
ADD	This will **add (append)** an item to the **end** of the collection and updates the current position. You must provide the new item data as an argument.
DELETE	This will **remove** the **current item** from the collection.
REPLACE_AT	This will **replace** an item in the collection at the specified **position** (zero-based) and updates the current position. You must provide the new item data as an argument, and the position.
DELETE_AT	This will **remove** an item at the **position** (zero-based) specified and updates the current position. You must provide the position as an argument.

> **NOTE**
>
> To access these actions (functions), you will need to apply the dot `.` operator of the collection variable.

Collection Declaration

Declaring a collection variable will require you to note it is a **collection type** and must **always initialize it to empty**. This is a good practice but also simplifies how we make the declaration. If we need to create a collection for storing many student ID's, we would declare as follows:

> studentIDs = EMPTY COLLECTION

This notation makes it clear the variable is called `studentIDs` and that it's a collection given it's being set to an **EMPTY COLLECTION**.

Example Scenario

It is very common for an application to manage information/data that supports many values which may or may not have a known limit to the number of pieces of data to maintain. For example, if an application is required to manage student ID's, we would need the application to be able to adapt to the number of student ID data (which could also involve the adding and removing of students as part of the features of the application).

We certainly would not want to represent each student ID number as a separate variable (such as: `id1`, `id2`, `id3`, etc..) since we would not know how many variables to declare in advance and how would we be able to add more at runtime?

Ideally, we need a way to simplify how we can represent data as a **list**. An important feature for this type of data representation would also include the ability to adapt to changes in the number of list items (expanding or contracting in size). This is where the concept of a collection comes in!

ONE VARIABLE!

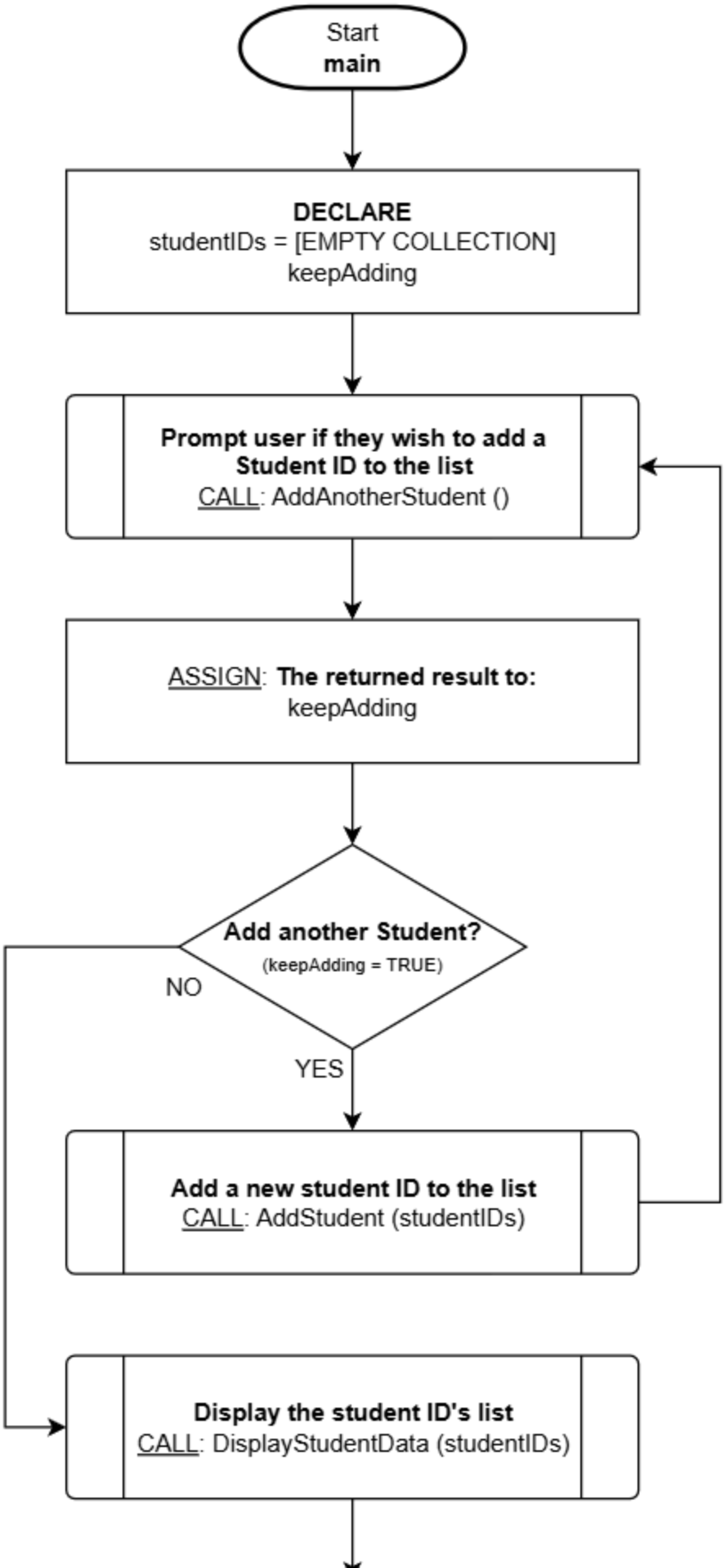
Instead of declaring MANY variables for each student ID (and given the impossibility of how we could even manage varying number of students), we can use a **SINGLE collection variable** to represent many student ID's (as a list)!

We can declare a **single collection variable**: `studentIDs` and by referring to this one variable, we can **access many student ID's**.

Let's define the process for an application to manage the creation and adding of student ID's to a collection variable (where the user can enter as many ID's as desired) with the functionality to display the data stored in the collection after the data is input. We will use a flowchart to orchestrate the main flow and pseudocode to describe the detailed parts accordingly.

Flowchart

This is the main function and logic flow.


Start
main
DECLARE
studentIDs = [EMPTY COLLECTION]
keepAdding
Prompt user if they wish to add a Student ID to the list
CALL: AddAnotherStudent ()
ASSIGN: The returned result to:
keepAdding
Add another Student?
(keepAdding = TRUE)
NO
YES
Add a new student ID to the list
CALL: AddStudent (studentIDs)
Display the student ID's list
CALL: DisplayStudentData (studentIDs)

Pseudocode

There are three functions to be detailed in this example.

Function: `AddAnotherStudent()`

```
------------------------------------------------------------------
Description/Purpose:
Prompt the user to confirm if another student ID needs to be
entered.

Argument(s) : **NONE**
Return Value: TRUE for "YES"
              FALSE for "NO"
------------------------------------------------------------------

AddAnotherStudent()

1. DECLARE:
      response = FALSE

2. DISPLAY:
      "Do you want to add a student ID?"
      [BUTTON: YES]       [BUTTON: NO]

3. What button was pressed?
      A) YES:
            1) ASSIGN: response = TRUE
      B) NO:
            1) ASSIGN: response = FALSE

4. RETURN: response
```

Function: `AddStudent (studentIDs)`

```
------------------------------------------------------------------
Description/Purpose:
Prompt the user to enter the new student ID and add it to the
collection.

Argument(s) : studentIDs (collection variable)
Return Value: Nothing
------------------------------------------------------------------

AddStudent (studentIDs)

1. DECLARE:
      newStudentID

2. DISPLAY:
      "Enter a new student ID:"

3. ASSIGN: newStudentID = [User entered value]

4. studentIDs.ADD(newStudentID)

5. DISPLAY:
      "Student ID added!"

6. End
```

Function: `DisplayStudentData (studentIDs)`

```
------------------------------------------------------------------
Description/Purpose:
```

Data Structures

Overview

Data structures are an extension of variable types. This is a powerful way to **group related information into a new data type**.

For example, if we want to represent a Seneca Student, we could group all the closely related attributes together into one variable type and make a variable of this new type which would contain all that related information in a single variable. For this example, let's say we want to manage the student ID, program code, and their graduation status.

Structure DEFINITION:

To define a new data structure, it starts with the new type name and followed by a list of attributes that comprise the new type.

```
[New Data Structure TYPE name]
 - [Attribute 1]
 - [Attribute 2]
 - [Attribute 3]
 - etc.
```

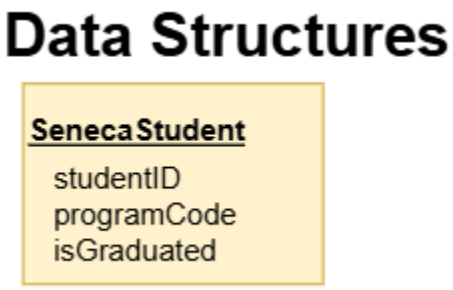
Here's how the Seneca Student type would be defined.

Pseudocode:

```
SenecaStudent
```

Flowchart:

All data structures would be grouped together in their own dedicated section with each data structure in a separate rectangle.



Creating a new data **TYPE** called `SenecaStudent` will enable us to represent these three attributes of data in a single variable (the student ID, program code, and the graduation status of the student).

> **NOTE**
>
> The structure **DEFINITION is NOT a variable** - it is only **describing** the attributes that can be managed (represented) for this new type.

Structure VARIABLE

As mentioned back in the variables topic, data **type** information (example: whole number, fractional number, characters etc.) is not supported in all programming languages and so we are maintaining the language agnostic approach and not specifying this level of detail when we declare variables.

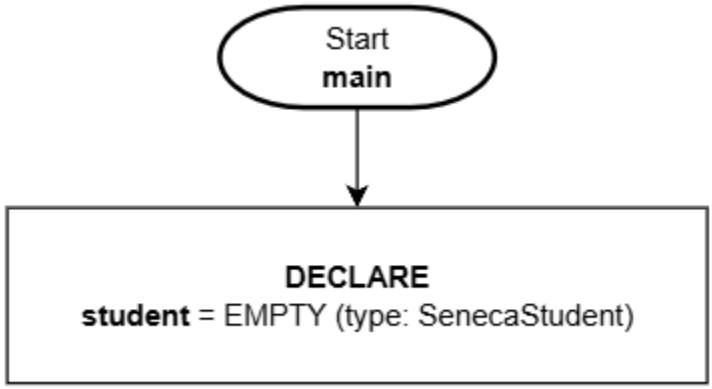
The exception to this rule is for data structures. To be able to USE a data structure (based on its definition), we must create a variable **including the type** so it will be clear it is not a standard variable.

Using the `SenecaStudent` structure example, this is how we can declare a variable of type **SenecaStudent**:

Pseudocode:

```
DECLARE:
      student = EMPTY (type: SenecaStudent)
```

Flowchart:



Now that we have a variable called `student` of type (`SenecaStudent`), we will be able to refer to and assign three related pieces of data using this **single variable**.

Access Structure Data

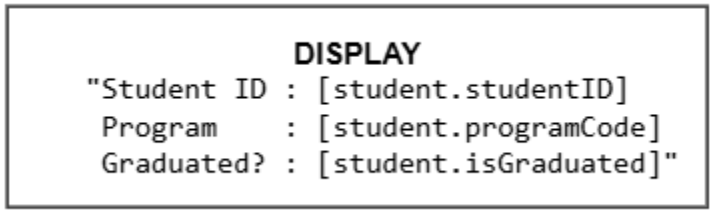
Once you have a variable declared that is a structure type, you will need to provide some extra syntax (`.`) to access or refer to the attributes.

Accessing the attributes for the variable `student` from our previous example (recall, `student` is of type **SenecaStudent**), here is how we would display the three attributes.

Pseudocode:

```
DISPLAY:
      "Student ID: [student.studentID]
       Program   : [student.programCode]
       Graduated?: [student.isGraduated]"
```

Flowchart:



> **FYI**
>
> The dot (`.`) syntax is very common among most programming languages to access sub-components of an object.

Structure COMPOSITION

Often, we need to include other defined structured data within another structure as there may be a loosely associated relationship involved. Let's revisit the previous example to demonstrate what this means.

Every student will have name information associated to their student ID. Names can be be simple, but they can also have more structure to them such as:

- First name

- Last name
- Middle name
- Nickname (short version of the name)
- Prefix (ie: Mx., Mr., Mrs., Ms., Master, etc.)

We can update our student example now to include this data representation:

Pseudocode:

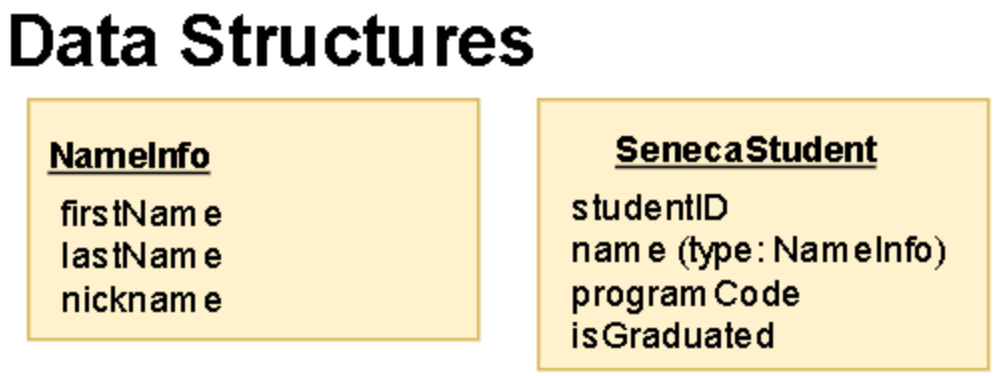
```
NameInfo
- firstName
- lastName
- nickname (short version of the name)
```

Then add this to our SenecaStudent data structure as a member:

```
SenecaStudent
- studentID
- name (type: NameInfo)
- programCode
- isGraduated
```

> **NOTE**
>
> Notice the additional attribute after `studentID`? We explicitly identify the `name` attribute as being of a special **NameInfo** type.

Flowchart:

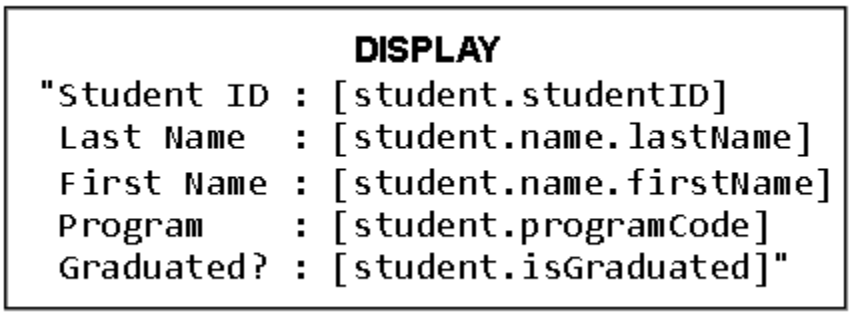


Accessing the details of the name follows the same convention as before, only now we will need to "drill" down into the data using the `.` dot notation when we get to that attribute. Here's how we would display the first and last name of a student with this new change:

Pseudocode:

```
DISPLAY:
      "Student ID: [student.studentID]
       Last Name : [student.name.lastName]
       First Name: [student.name.firstName]
       Program   : [student.programCode]
       Graduated?: [student.isGraduated]"
```

Flowchart:



> **NOTE**
>
> Notice the extra level of using the dot (`.`) to access the attributes of the `name`? Since `name` is a structure, we need to access its attributes using the `.` notation.

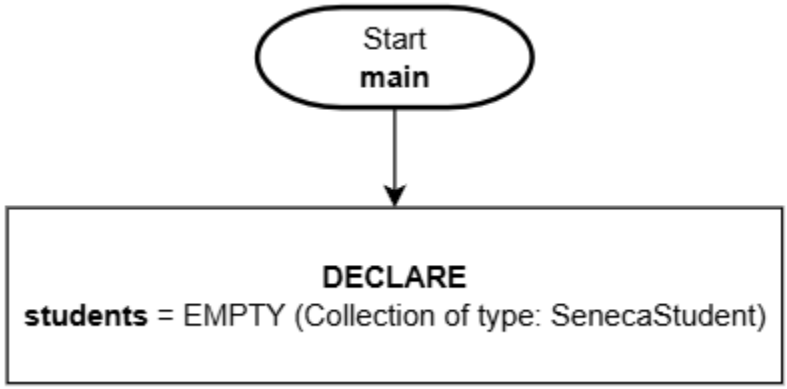
Collection of Structures

Now that we have the ability to create structures to better contain related data - let's look at how you can have a collection of structures. In the above examples, we have a single student represented in a variable named "student". What if we want to manage many students? Just as we did in the introduction to collections, we can specify a collection of a new type `SenecaStudent`.

Pseudocode:

```
DECLARE:
      students = EMPTY (Collection of type: SenecaStudent)
```

Flowchart:



Example - Putting it all together!

Here is an example of putting it all together where we will support **adding, removing, and displaying** the data for a collection of `SenecaStudent` data.

Flowchart

This is the main function and logic flow.

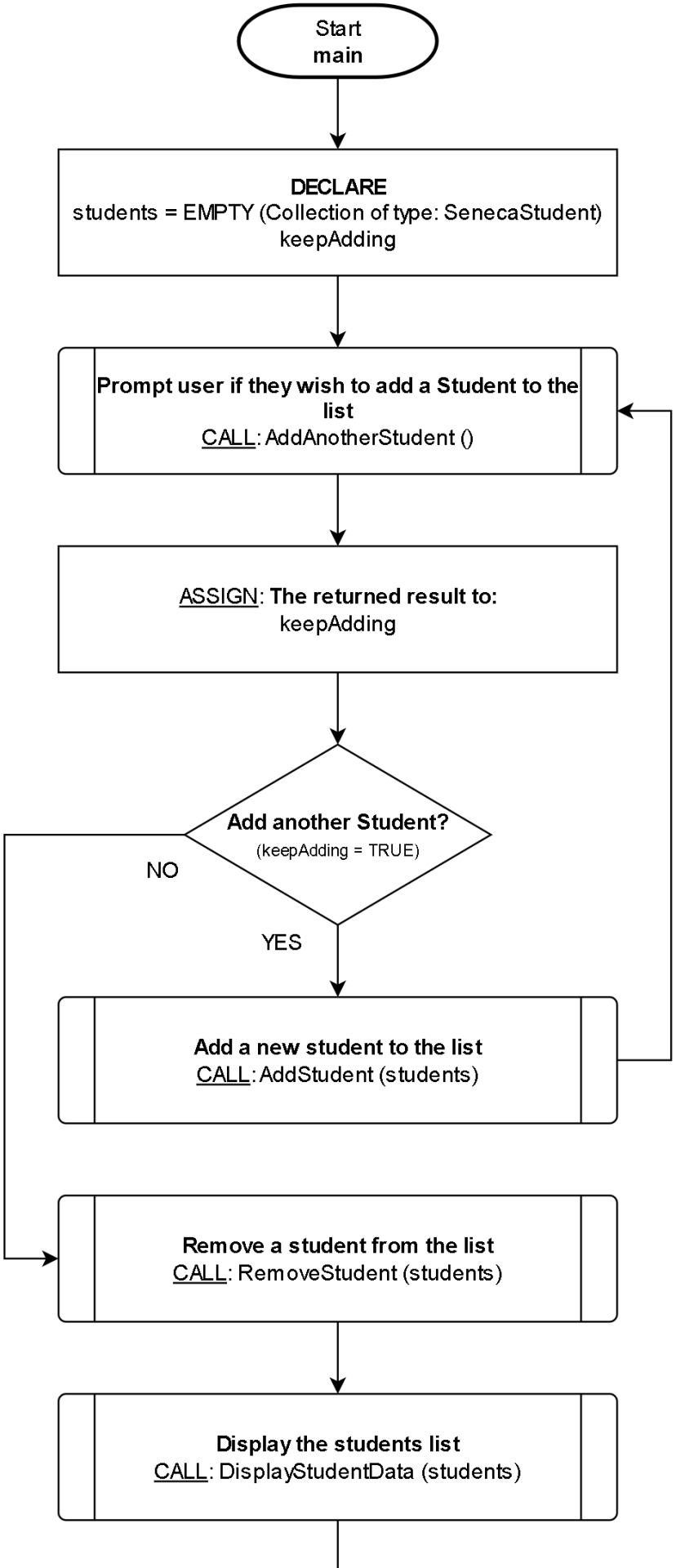
Data Structures

NameInfo

firstName
lastName
nickname

SenecaStudent

studentID
name (type: NameInfo)
programCode
isGraduated



Pseudocode

First we will define the `SenecaStudent` data structure and then the four detailed functions needed in this example.

```
-----------------------------------------------------------------
Data Structures:

NameInfo
- firstName
- lastName
- nickname (short version of the name)

SenecaStudent
- studentID
- name (type: NameInfo)
- programCode
- isGraduated

-----------------------------------------------------------------
```

Function: `AddAnotherStudent()`

```
-----------------------------------------------------------------
Description/Purpose:
Prompt the user to confirm if another student needs to be entered.

Argument(s) : **NONE**
Return Value: TRUE for "YES"
              FALSE for "NO"
```

Function: `AddStudent (students)`

```
-----------------------------------------------------------------
Description/Purpose:
Prompt the user to enter the new student data and add it to
the collection.

Argument(s) : students (Collection of type: SenecaStudent)
Return Value: Nothing
-----------------------------------------------------------------

AddStudent (students)

1. DECLARE:
      newStudent (Type: SenecaStudent)

2. DISPLAY:
      "Enter a student ID:"

3. ASSIGN: newStudent.studentID = [User entered value]

4. DISPLAY:
      "Enter the first name:"

5. ASSIGN: newStudent.name.firstName = [User entered value]

6. DISPLAY:
      "Enter the last name:"

7. ASSIGN: newStudent.name.lastName = [User entered value]

8. DISPLAY:
      "Enter the program code:"
```

Function: `RemoveStudentData (students)`

```
-----------------------------------------------------------------
Description/Purpose:
Removes a student from the students collection where it matches
on the user-entered student ID.

Argument(s) : students (Collection of type: SenecaStudent)
Return Value: Nothing

-----------------------------------------------------------------

RemoveStudentData (students)

1. DECLARE:
      tmpStudent
      tmpID
      foundMatch = FALSE
      keepLooking = TRUE

2. DISPLAY:
      "Enter the studentID of the student to remove:"

3. ASSIGN: tmpID = [User entered value]

4. Continue searching the collection? (keepLooking = TRUE)
      A) YES:
            1. ASSIGN: tmpStudent = students.NEXT
            2. Is tmpStudent empty?
                  A) NO:
                        1. Does the entered ID match this
student? (tmpStudent.StudentID = tmpID)
                              A) YES:
                                    1. REMOVE the current student
```

Function: `DisplayStudentData (students)`

```
------------------------------------------------------------------
Description/Purpose:
Display each student in the students collection (iterate/loop).

Argument(s) : students (Collection of type: SenecaStudent)
Return Value: Nothing

------------------------------------------------------------------

DisplayStudentData (students)

1. DECLARE:
      tmpStudent

2. DISPLAY:
      "Here are the stored students:"

3. ASSIGN: tmpStudent = students.NEXT

4. Is tmpStudent empty?
      A) NO:
            1. DISPLAY:
                  "Student ID: [student.studentID]
                  Last Name : [student.name.lastName]
                  First Name: [student.name.firstName]
                  Program   : [student.programCode]
                  Graduated?: [student.isGraduated]" (newline)
            2. REPEAT: from Step #3

5. DISPLAY:
      "There are [students.COUNT] Student's stored."

6. End
```

Date and Time

Overview

Working with date and time data can be very tedious depending on how the data is needed and used in a problem. Almost all programming languages have supporting libraries to help simplify how we work with this type of information. Maintaining the language agnostic theme of these notes, this appendix will set the framework for how date and time data can be used.

CURRENT DateTime Constants

It is common for programmers to access the **current** date and time information in solutions. You will need to refer to the keyword "`NOW`" to access the different attributes of the date and time parts. Below is a table showing all the parts and how you would access specific date and time parts:

Date/Time Part	Result
`NOW`	2025-06-01 23:59:59
`NOW::Date`	2025-06-01
`NOW::Time`	23:59:59
`NOW::Year`	2025
`NOW::Month`	06 OR JUNE or JUN

Date/Time Part	Result
`NOW::Day`	01 OR MONDAY or MON...
`NOW::Hour`	23
`NOW::Minute`	59
`NOW::Second`	59

> **NOTE**
>
> `Month` and `Day` parts are **ABSTRACTED** meaning these can represent either the numerical or alpha representations (short or long form). This is for flexibility based on the context in which it is used so it is **important your logic and interface refer to what form these values will be used**.

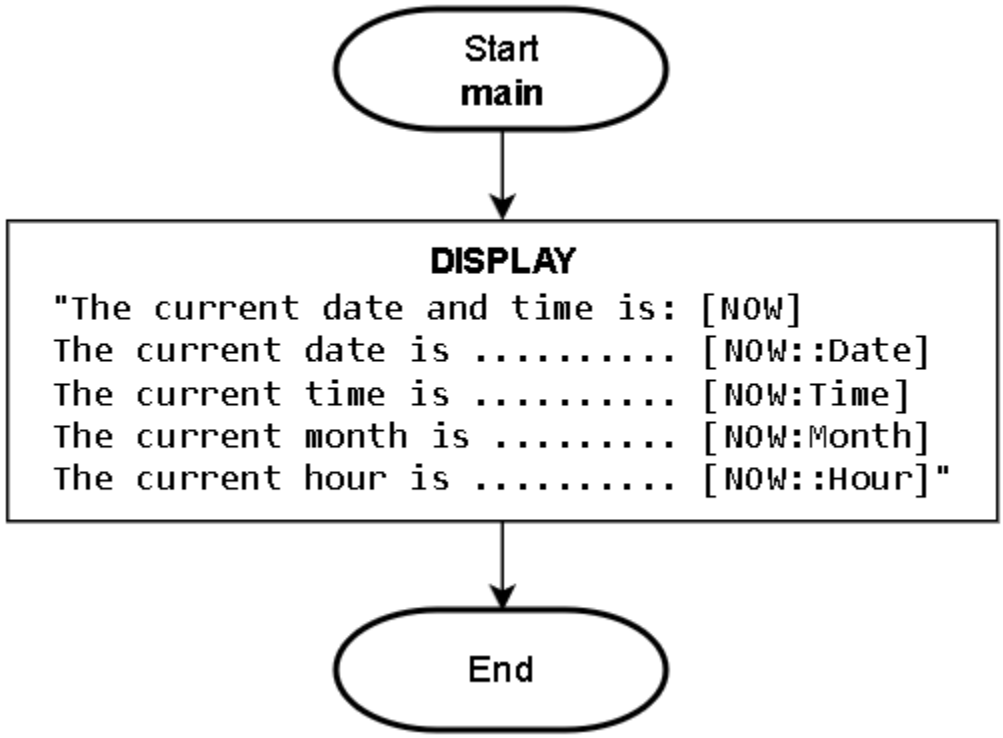
Example

Displaying some CURRENT date and time parts would be done like this:

Pseudocode:

```
1. DISPLAY:
                "The current date and time is: [NOW]
                 The current date is ......... [NOW::Date]
                 The current time is ......... [NOW:Time]
                 The current month is ........ [NOW:Month]
                 The current hour is ......... [NOW::Hour]"
```

Flowchart:



VARIABLE of Date and Time

In addition to the **CURRENT** date and time, we often need to **STORE** date and time data which will require the use of a variable. Using a variable to access various parts of the date and time data, is identical to the preceding CURRENT date and time section only instead of using `NOW`, the declared `variable` will be applied:

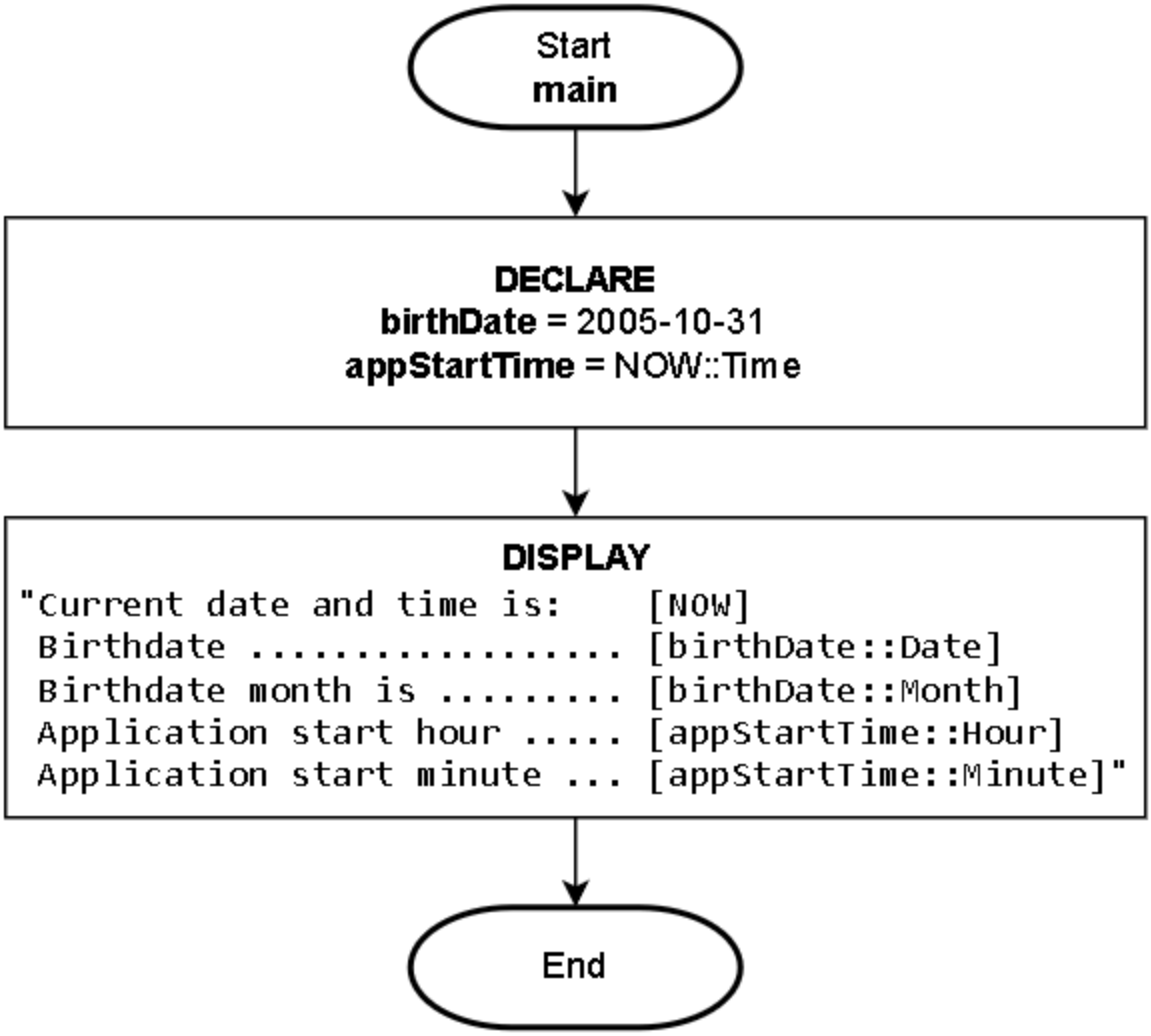
Example

Displaying some VARIABLE date and time parts would be done like this:

Pseudocode:

```
1. DECLARE:
                birthDate = 2005-10-31
                appStartTime = NOW::Time

2. DISPLAY:
                "The current date and time is: [NOW]
                 Birthdate ................... [birthDate::Date]
                 Birthdate month is .......... [birthDate::Month]
                 Application start hour ......
[appStartTime::Hour]
                 Application start minute ....
[appStartTime::Minute]"

3. End
```

Flowchart:


Start
main
DECLARE
birthDate = 2005-10-31
appStartTime = NOW::Time
DISPLAY
"Current date and time is: [NOW]
Birthdate ..... [birthDate::Date]
Birthdate month is ..... [birthDate::Month]
Application start hour ..... [appStartTime::Hour]
Application start minute ... [appStartTime::Minute]"
End

Timer Logic

Overview

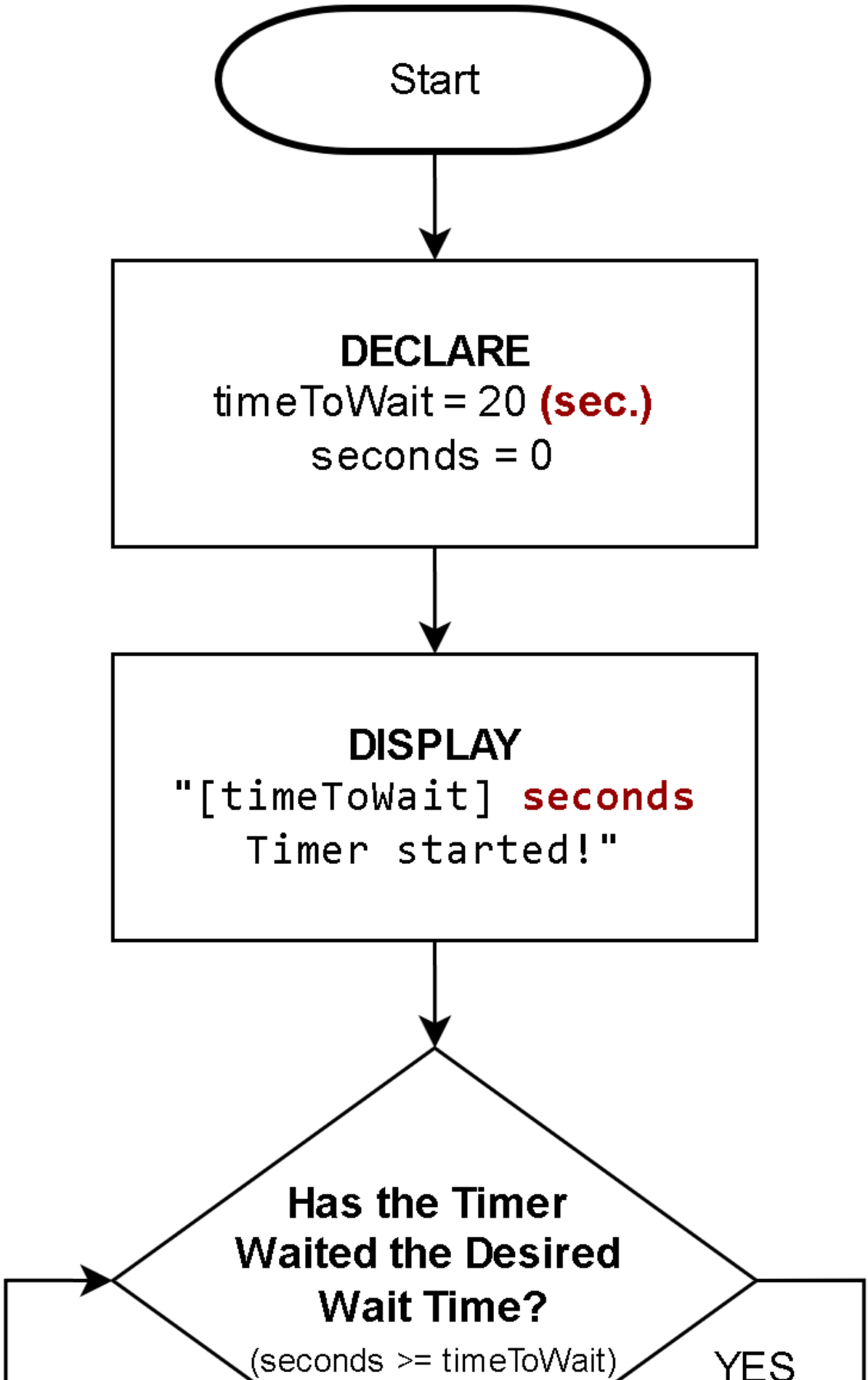
Timers are often needed in programming. There are cases when accuracy is very important and other cases where approximation is good enough. This section will be focused on how to accurately apply timer logic.

APPROXIMATE Timer

Many programming languages have libraries to help simplify how time is tracked and implemented in programs. Among the many features, is usually a function that can be used to `wait` or `sleep` for a specified duration (an argument would be sent usually in unit seconds or milliseconds). However, such a function is not an accurate implementation of time and is more an approximation. Here's one such example of how it would be used in a NON-CRITICAL timer for 20 seconds:

Example

Flowchart


Start
DECLARE
timeToWait = 20 (sec.)
seconds = 0
DISPLAY
"[timeToWait] seconds
Timer started!"
Has the Timer
Waited the Desired
Wait Time?
(seconds >= timeToWait)
YES

Pseudocode

```
1. DECLARE:
        timeToWait = 20 (seconds)
        seconds = 0

2. DISPLAY:
        "[timeToWait] seconds Timer started!"

3. Have we waited the desired time?
        A. NO (seconds < timeToWait):
                        1. ASSIGN: seconds = seconds + 1
                        2. CALL: wait(1)    <=== this will pause
the application for 1 second
                        3. REPEAT: from step #3

4. DISPLAY:
                "[timeToWait] seconds is up!"

5. End
```

This example is not accurate because the timer piece of waiting is embedded among other logic (ie: the loop) that takes time to execute in it's own right. Iterating this loop for longer times, will increasingly introduce inaccuracy and actually take longer (more seconds)! There are other factors influencing the inaccuracy as well which ties back to the CPU's architecture (ie: step logic), system power state (ie: laptop on low power vs. performance mode) and other things. So, how can we implement this to be accurate?

ACCURATE Timer

To accurately apply timers, we should use **actual time** itself! By applying a "stop watch" logical approach, we can accommodate for outside factors that

otherwise influence the inaccuracies of the library timer approach.

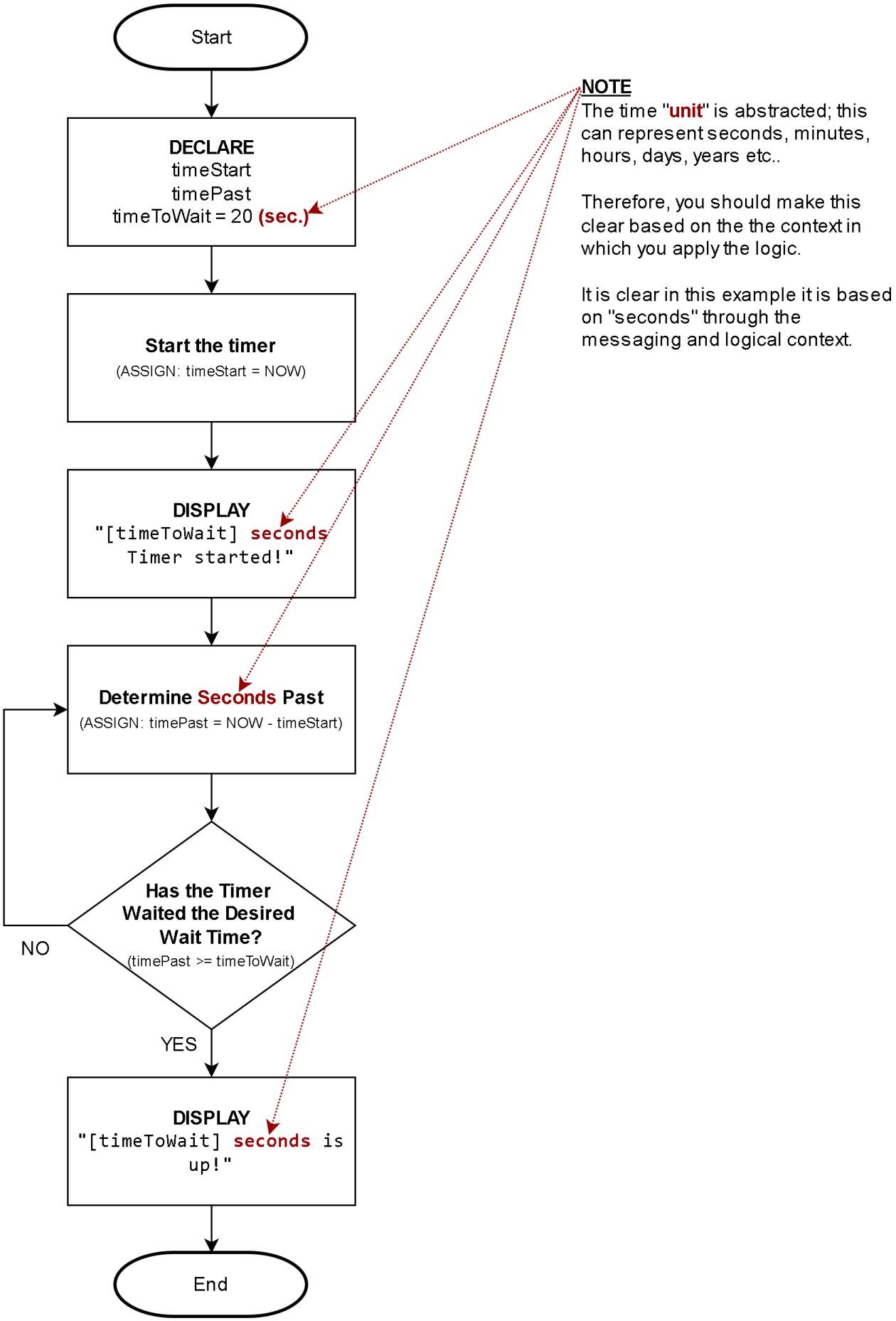
The first thing we need to do is note the timer start-time which would be set to NOW, then periodically, perform a calculation to determine the time passed since the timer started (based on subtracting the start-time from the current-time). The only influencing factor in maintaining the accuracy in this approach is how often you decide to check for the time passed. The other advantage of this is the unit itself, it is not limited to a programming language library function's unit of measure (usually in seconds or milliseconds) because we can logically test the time duration based on any part of time we wish (ie: hours, days, years)!

> **IMPORTANT**
>
> The time "unit" is abstracted; this can represent seconds, minutes, hours, days, years etc.. Therefore, you should make this clear based on the the context in which you apply the logic. In the following example, it is clear the unit is based on "seconds" through the messaging and logical context.

Example

Here is how you would set-up an **ACCURATE** 20-second timer:

Flowchart:


Start
DECLARE
timeStart
timePast
timeToWait = 20 (sec.)
Start the timer
(ASSIGN: timeStart = NOW)
DISPLAY
"[timeToWait] seconds
Timer started!"
Determine Seconds Past
(ASSIGN: timePast = NOW - timeStart)
Has the Timer
Waited the Desired
Wait Time?
(timePast >= timeToWait)
NO
YES
DISPLAY
"[timeToWait] seconds is
up!"
End
NOTE
The time "unit" is abstracted; this can represent seconds, minutes, hours, days, years etc..
Therefore, you should make this clear based on the the context in which you apply the logic.
It is clear in this example it is based on "seconds" through the messaging and logical context.

Pseudocode:

```
1. DECLARE:
        timeToWait = 20 (seconds)
        timeStart
        timePast = 0

2. ASSIGN: timeStart = NOW

3. DISPLAY:
               "[timeToWait] seconds Timer started!"

4. ASSIGN: timePast = (NOW - timeStart) <-- NOTE: The context is
in sec.,
                                                    so the result
is in seconds

5. Have we waited the desired time?
        A. NO (timePast < timeToWait):
                        1. REPEAT: from step #4

6. DISPLAY:
               "[timeToWait] seconds is up!"

7. End
```

Glossary of Key Terms

Collection

Collections are composites of other objects that include data and have logical functionality to support the containers operations.

Construct

A logical control such as a selection or iteration.

Container

Containers are very data specific in the simplest forms such as an array and do not inherently have a lot of logical functionality.

Enumeration

Sequencing steps with an identifier such as a number (ex: 1. 2. 3. ...) or letter (ex: A. B. C. ...).

Used primarily in pseudocode to identify the execution sequence of statements.

Indentation

A fixed number of blank spaces to the right (ex: using the `TAB` key) used for aligning grouped logic that is nested.

Iteration

A construct that enables looping in a program - the ability to repeat a statement(s) easily.

Jump Statements

A poor logic flow that directs the next step of execution be an arbitrary location outside of the immediate vicinity of the current statement (ex: go to step #152). Instead, logic should be controlled using logical constructs such as selection and iteration.

Modularity

A term used for breaking down logic into various degrees of detail. This can be at a directory-level, file-level, function-level, or statement-level.

Nest

Content (statement(s), selection, or iteration) residing within a preceding construct such as a selection or iteration.

The nested content would be dependant on a preceding construct and in the case of pseudocode would be indented under it.

Obfuscate

To purposely confuse and misdirect meaning usually by way of applying meaningless names to variables and functions.

Selection

A construct that enables decisions in a program to adapt to changing conditions and execute different logic flows.

Semantic

Is another word for logic. It is the logical definition and meaning of an algorithm.

Statement

A single line of instruction (step) such as a calculation, action to display or receive user input.

Variable

A named placeholder used to reference a value. These are used for storing data for later use and evaluation.