Chapter 2 Recap

Please note that the following are mainly points with some brief explanations to add to the notes you are compiling. The recap is an opportunity to revisit some of the work we have spoken about.

Overview of Computers and Programming

Chapter highlights:
-- Declaring variables and constants
-- Performing arithmetic operations
-- The advantages of modularization
-- Modularizing a program
-- Hierarchy charts
-- Features of good program design

Declaring variables and constants

- Programming is and will always be the opportunity to manipulate values and given the chance, manipulate Storage on a disk because of a task that needs to be performed.
- This is not a textbook definition - so it’s important to research the right definitions when it is needed

-- **Variables and Literal Constants**

- when declared in a program, a computer proceeds to set up a specific memory location to hold the values of a variable and a constant. They are identified by a name and, a data type needs to be known since the computer needs to know how much of memory to allocate to the variable or constant being declared.

-- *Variables*

- A variable is a value that may change during the processing of a program
- Some environments call variables, identifiers

-- *Constants*

- A constant is a value that never changes during the processing of all the instructions in a program.
- They can be any data type
- Example – PI is always a constant value

```
const float PI = 3.14; // a code example in a programming language
```

- The above variable will be used as required and will never change.
- If there is code that attempts to change the value, the compiler/interpreter will present an error stating that this is not allowed.

- *Below is an explanation of using unnamed and literal constants*

-- Unnamed vs Literal Constants

- Named Constant
-- It is a variable, and the value is assigned at declaration and only once
-- The value cannot be changed during the execution of a program
-- Example – VAT is the constant variable - `fltNewPrice = fltPrice + (fltPrice * fltVAT)`

- Literal Constant
-- Unnamed constant
-- The value that is used in a program – It is hard-coded
-- Example – The VAT value is used - `fltNewPrice = fltPrice + (fltPrice * 0.15)`

-- **Data Types**

-- Classification describes the type of value that can be stored in the declared variable

- **Numeric**
  - Numbers
  - This type can be used to perform mathematical operations
  - **-- Integers**
    - All whole numbers - both positive & negative (1024, -256)
  - -- **Floating-point**
    - All real numbers - both positive & negative (-9876.55, 34522.0, 0.0003453)
- **String**
  - Consists of alphanumeric characters
  - Character and Strings variables cannot be used for mathematical operations, however they can be compared and arranged in alphabetical order.

  -- Each character is assigned a numerical value based on the ASCII chart (see the ASCII chart - https://www.ascii-code.com/) - we read the decimal number to understand this concept, but it is important to recognize that the computer will retrieve the binary number equivalent in order to transfer information internally.

  -- When we join multiple characters, we are actually forming a string

  -- To join more than one string or character to form a new string is known as concatenation, using the + symbol in coding
  -- Example - "5" + "5" will result in "55" and not 10
  -- **Character**
    - All letters (CAPITAL and small), numbers, and special symbols ('A', 'h', '5', '&')
  -- **String**
    - Combination of more than one character ("More than one word", "ST99999", "7676", "76 & 32")
- **Logical / Boolean**
  - only able to contain either the value: True, or the value: False

- programmer designates the data type during the programming process
- the program then will recognize the name and know the assigned data type

- Data types cannot be mixed
- **Type safety**: Prevents assigning values of an incorrect data type into a variable
- *If a number value is not going to be used in a mathematical equation or expression, then it should be declared as a string*

-- **Declaration / Identifier**

- Remember that when a variable or constant has been declared, then the computer will store the data internally in a named memory location. You are able to access this memory location by stating the name of the variable when needed.

- *In the case of a variable, there is only one value that is stored in the memory location, but this value may be overwritten by another, since the theory we have read earlier, states that a variable can change during the execution of a program.*

- *In the case of a constant, there is again, only one value store in the memory location, but this value may not change at any time during the execution of a program.*

-- Working with variables
- Declaration statement
  (type & identifier) `num intNumber`
- Initializing a variable
  This can be done in 2 ways.
  - at declaration `num intCount = 0`
  - or it can be done after declaration on a different line

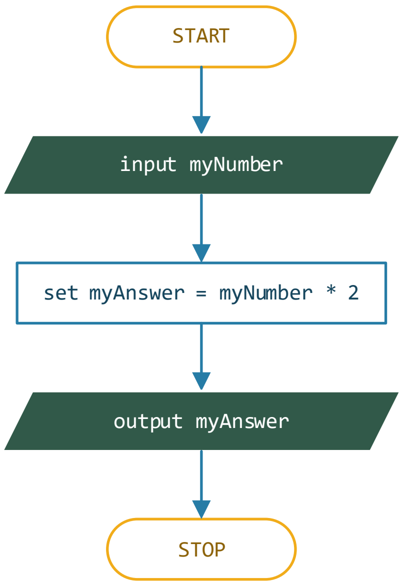
```
num intCount
intCount = 0
```

-- Choosing to initialize a variable with a value will be your choice, unless specifically requested
-- The idea of setting a value, helps with understanding how the variable is likely to be used and what kind of value it is going to begin with

Pseudocode

```
start
        input myNumber
        set myAnswer = myNumber * 2
        output myAnswer
stop
```

Flowchart

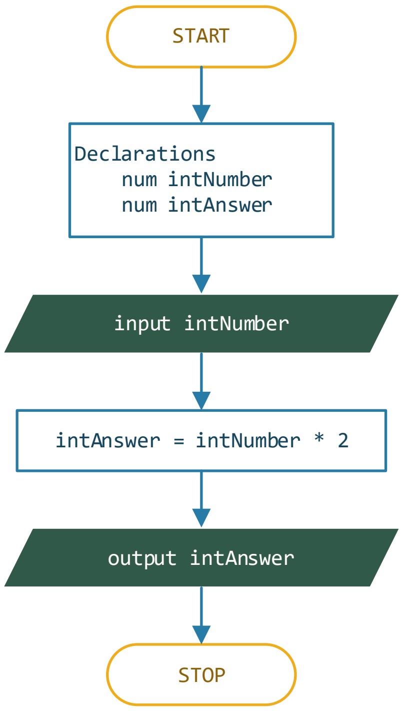


Edit of the above code - now with the inclusion of the declarations & the standard we will be using for our classes.

Pseudocode

```
0. Start
1. Declarations
        num intNumber
        num intAnswer

2.      input intNumber
3.      intAnswer = intNumber * 2
4.      output intAnswer

5. Stop
```

Flowchart



-- **Naming Variables**

- When you need to name variables, you will need to follow the naming conventions of the company for which you are working. It is important for all programmers within a company to follow the convention so that there is a consistency in all the coding of your programs.

- In the future when there are many programmers working on one solution, following the naming conventions will eliminate conflicting names. It will also be readable amongst the team and the company. Also allows for easy maintenance of the program. Naming conventions produce clean, well-written programs.

- Following the information about the data types in a previous section, we will now use the following naming convention in all our programs. This is the standard we are following in our classes in order to maintain a standard amongst the programs that are shared.

```
num intNumber
num fltPrice
// num is the classification that this is a numerical variable

string chrOneCharacter
string strCompanyName
// string is the classification that this is an alphanumeric variable

boolean blnCheck
// boolean is the classification that this is a boolean variable that
// only could have a value of true or false
```

-- The following is the current list of prefixes that will be used along with the names of variables and constants

- integer	**int**
- float	**flt**
- character	**chr**
- string	**str**
- boolean	**bln**

*** more prefixes will be added to this list as we learn more data types that exist in programming*

- Name a variable according to what it represents
- *Is it descriptive enough?*

- using one word is acceptable - fltPrice - because we have an understanding that the value being stored in memory is the price of an item

- consider one word, but if there is more than one word, each word must start with a capital letter
-- e.g., fltHourlyWage, intRandomNumber

- Names are case sensitive
- ** **RESERVED words** are not allowed to be used as names of variables or constants
- Do not start a name with a digit/number - always start with a letter
- Do not name variables with spaces in between words
- For names of constants - please use CAPITAL LETTERS - fltVAT or fltPI

-- **Assigning Values to Variables**

- Assignment statement
-- `intAnswer = intNumber * 2`
-- the variable on the left is assigned the resulting value of the expression on the right-hand side of the equal to operator

----
Performing arithmetic operations
===

While the above heading and slides, provides a basic understanding of the arithmetic operations allowed, it would be a little better to understand the concept of operators in programming as a whole and learn as much as we can, so that we better understand future sections. Please note that the following notes is taken from another textbook and based on my own knowledge.

Functions
===

** **Functions** (*something a little different in comparison to the discussion below about Modules, but there are similarities we will see moving forward*)

- These are small sets of instructions that perform a task and return a value.
- They are usually built into the programming language
- The syntax of calling a function - FunctionName(data)

- Since a function will return a value, most of the time, we will consider a variable to accept the value returned
- Example: fltResult = SQRT (10)
// the result of the square root of 10 is assigned to the variable, fltResult

- The code: (data), mentioned above is referred to as a "parameter" and some others refer to it as the "argument that is passed", in this case to the function
- There are many pre-loaded functions that are available to use, even within the scope of the pseudocode we are learning, and we will learn then as we come across them

- There are few types of functions, and I will highlight them below:

-- **Mathematical functions**
 - used in science or business, calculating, square root absolute value, random number and much more

-- **String functions**
 - these are used to manipulate string variables. We can copy parts of a string to another variable, we can find the length or a specific character, and again, much more ...

-- **Conversion functions**
 - these are used to convert data from one data type to another. Not used much in pseudocode, but in programming languages, we will see a need for them.

 - Example, in *Java*, there is function/window that we can use to read in a value, **JOptionPane**, but when a value is entered and returned from the function, it is always a value of the String data type and if we asked for a number, we would need to use a conversion function to change the string value (a list of characters) into a number

-- **Statistical functions**
 - used to calculate maximum and minimum values, mean and median values, and so forth

-- **Utility functions**
 - When we need to produce reports, for example, utility functions will be used. These functions can access information outside of the program and the language of the PC.
 - For example, we can retrieve the data and time and be able to display it in a report when required.

Operators

===

-- Operators are data connectors within expressions and equations. They help us tell a program how to process the data. The type of operator will also help us understand the type of data that can be processed and the value that likely to be returned from the execution of the expression or the equation.

-- 3 Categories of Operators

- **Mathematical**
    - -- We've dealt with mathematical operators for all throughout schooling, so we have an understanding for them.
    - -- The extra bit, is the addition of *integer division and modulo division*

- **Relational**
    - -- A programmer would use relational operators to program decisions

- **Logical**
    - -- Used to connect relational expressions (decision-making expressions) and to perform operations on logical data

- **List of Operators that will be used in our programs**

-- Mathematical

^	To the Power of
*	Multiplication
/	Division
\	Integer division (produces the integer part as the answer)
MOD	Modulo (or modulus) division (produces the remainder of the division as the answer Remember long division from school days)
+	addition
-	subtraction

-- Relational

<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to
=	equal to
<>	not equal to

-- Logical

NOT
AND
OR

-- **Order of Precedence** (Operators)

Operator	Operand	Result
()	Brackets ** Reorders the hierarchy; all operations are completed within the brackets using the same hierarchy	
	Functions – Internal functions based on the information provided in the previous section	
Mathematical		
^	Numeric	Numeric
\, MOD	Numeric	Numeric
*, /	Numeric	Numeric
+, -	Numeric	Numeric
Relational		
=, <, >, <=, >=, <>	*Numeric, Character or String*	Logical (True or False)
Logical		
NOT	Logical	Logical (True or False)
AND	Logical	Logical (True or False)
OR	Logical	Logical (True or False)

-- **Operands & Results**

-- The operand and the result are two ideas related to operators.
-- Operands are the data that the operator connects and processes.
-- The result is the answer that is produced when the operation is completed.

Example

```
fltPrice = fltPrice + (fltPrice * fltVAT)
fltWages = (40 * fltPayRate) + ((fltHours - 40) * fltPayRate * 1.5))
F = 6 * (2 \ 6 (6 + 2))
```

-- **Expressions and Equations**

-- Knowing the ideas and lessons around variables, constants and data types and not useful until you are able to use these concepts to create expressions and equations.

-- Now I would ask that you remember the previous lessons where we followed a story and then proceeded to experiment with values, and in-turn you had the opportunity to use the values that I had given as an example to produce an answer. This was you developing equations for the solution. After we developed the algorithm (*the program is shared below*), I trust the idea of an equation makes a little more sense now.

-- The problem you are trying to solve, is an opportunity for you to develop expressions and equations as you are required to - example - *calculate wages for an employee, include a tax and medical deduction - calculate the interest on a loan - sort a list of names in a class group - determine the highest and lowest number*

-- All these tasks require the use of different operators, and you would develop expressions and equations to perform the required tasks to arrive at the result

-- They make up part of the instructions ...

-- **Expressions**

- processes data, the operands, using operators
- Example - length * width
- Expressions do not use the equal to "=" sign - except as a relational operator
- The result is not stored and, therefore, unavailable for use at another time

-- **Equations**

- Stores the result of an expression in a memory location in the computer

(*The named variable being used will accept the answer*)

```
area = length * width
```

- * **AGAIN** - here in the assignment statement - the variable on the left is assigned the value of the result of the expression on the right-hand side

- The assignment statement intCount = intCount + 1 is quite interesting - we would not have learnt this during school days, but here in programming & based on how we read an assignment statement - *the variable intCount is assigned the result of the expression on the right which is, intCount + 1*

-- **Examples**

Expression	Equation
a + b	c = a + b
a < b	c = a < b
a AND b	c = a AND b

The advantages of modularization
===
- While this section is important to understand the creation of programs and in-turn the separation of code into smaller manageable blocks. Please note that this section will make more sense once we start to understand the development of algorithms/pseudocode beyond this chapter's notes.

- **Modules**
    -- sub-unit
    -- Other names we will be using, subroutines, methods, procedures, and functions
    -- Even though there are other names, the words do serve a purpose in how and when they are used

    -- When we wish to call a module, we use the name of the module to invoke it, thereby executing the code in the module as requested

    -- The process of breaking down a large program into modules (functional decomposition)
    -- Provides abstraction – a different type of thinking when dealing with and including modules in a program

    The following 2 statements is to be considered when you become a member of a team of developers.
    -- Becomes easier to divide the tasks among various people
    -- In this way, the team can then come together, and piece-by-piece put together a complete solution

- **Reusability**
    -- allows modules to be used in a variety of applications (API, Class Library)
- **Reliability**
    -- Assures that a module has been tested and proven to function correctly

----
Modularizing a program
===

- When we decide to include modules in a program, we are then made aware of some specific pieces of code that need to be used when coding and discussing a program

Consider the following program:

```
0. Start
1. Declarations
        num intMorningIn
        num intNoonOut
        num intNoonIn
        num intNightOut
        num intHours
        num intMins

2.      output "Please enter what time you come into work (in minutes since midnight)"
3.      input intMorningIn

4.      output "Please enter what time you left for lunch (in minutes since midnight)"
5.      input intNoonOut

6.      output "Please enter what time you came back from lunch (in minutes since midnight)"
7.      input intNoonIn

8.      output "Please enter what time you left for home (in minutes since midnight)"
9.      input intNightOut

10.     intHours = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) \ 60
11.     intMins = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) MOD 60

12.     output "The number of hours worked is " + intHours
13.     output "The number of minutes worked is " + intMins

14. Stop
```

- **Main Program**
    - -- The steps of the mainline logic of the program

- **Module**
    - -- Modules are invoked, at present, in the main program
    - -- We also use the phrase - "call a module”.
    - -- *future programs will be developed with modules being called where they are needed*

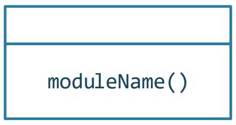
    -- A module has the following structure
    Module Header
    Module Body
    Module return statement.

    Example

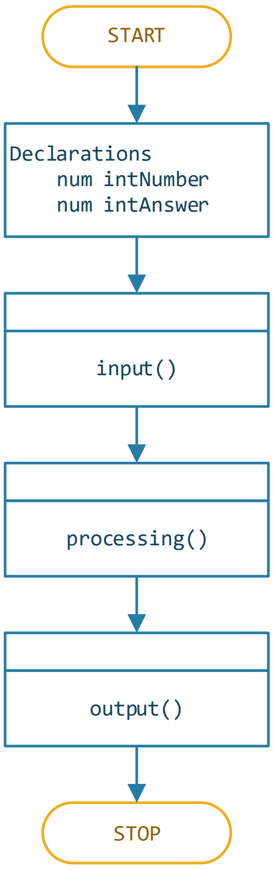
```
0. calcArea()
1.       intAnswer = intLength * intLength
2. return
```

    -- Naming a module is similar to the rules of naming a variable and should be descriptive once again
    -- All module names are followed by a pair of parentheses () / round brackets

- Flowchart
    - -- symbol / shape



-- Example
    -- Main Program calling procedures



- A concept that needs to be understood when dealing with Modules
    - statements that are taken out of a main program and put into a module have been encapsulated
    - Main program becomes shorter and easier to understand
    - *Again ** modules are reusable*

- **Functional cohesion**

    -- Occurs when elements of a module are grouped together because they are united for a single, well-defined purpose. All of the elements in the module work together to fulfil that purpose. Functional cohesion in a module is ideal and is the highest type of cohesion.

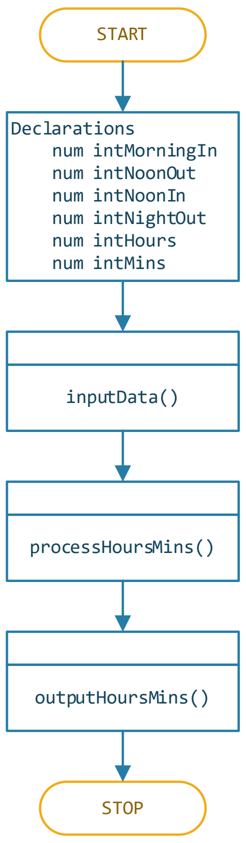
    -- *“Functional cohesion promotes the reusability of a module and makes it easier to maintain. Examples of functionally cohesive modules include one that is responsible for reading a particular file and one that is responsible for calculating shipping costs for an order.”* (Ingeno, J)

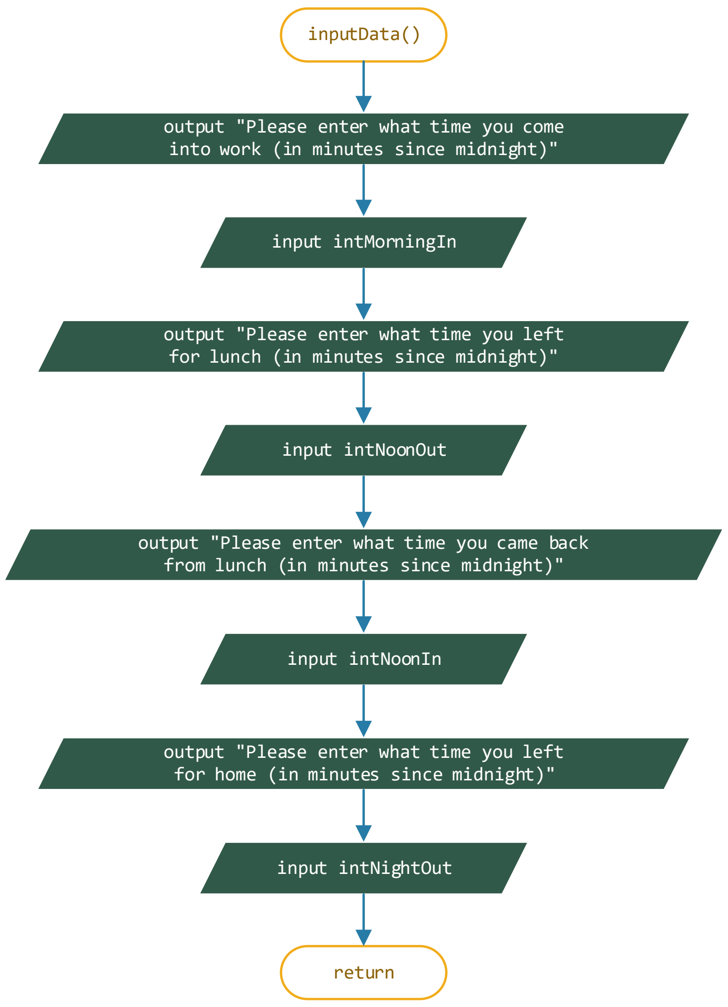
    (**Software Architect's Handbook by Joseph Ingeno** https://learning.oreilly.com/library/view/software-architects-handbook/9781788624060/?_gl=1*1vqxn1g*_ga*Mjk4NTQ1MDA2LjE2NzY0NjAxMTY.*_ga_092EL089CH*MTY3ODgwNDQwNi4yLjEuMTY3ODgwNDY2Ny42MC4wLjA.)

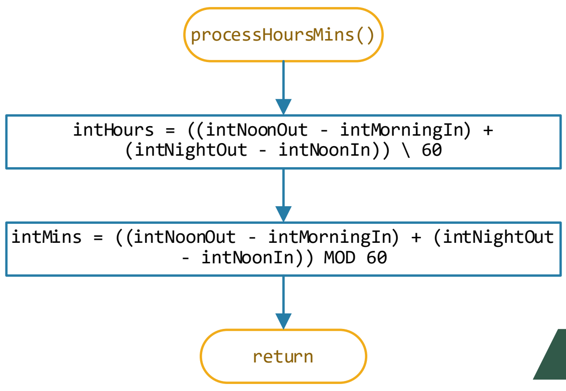
Now take the above code and please note down the how the idea of modules is used.

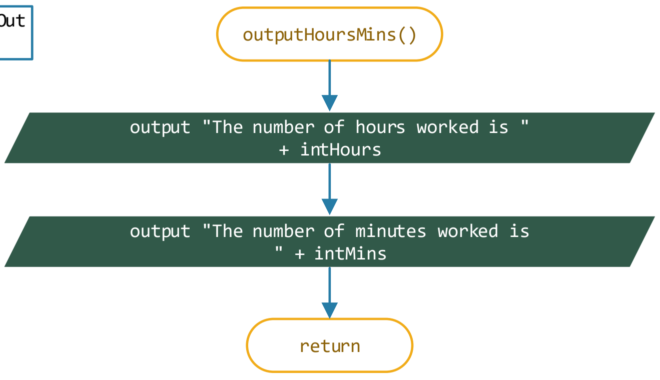
```
----------------------------------------------------------------------------------------------------
0. Start
1. Declarations
        num intMorningIn
        num intNoonOut
        num intNoonIn
        num intNightOut
        num intHours
        num intMins

2.      inputData()
3.      processHoursMins()
4.      outputHoursMins()

5. Stop
----------------------------------------------------------------------------------------------------
0. inputData()
1.      output "Please enter what time you come into work (in minutes since midnight)"
2.      input intMorningIn

3.      output "Please enter what time you left for lunch (in minutes since midnight)"
4.      input intNoonOut

5.      output "Please enter what time you came back from lunch (in minutes since midnight)"
6.      input intNoonIn

7.      output "Please enter what time you left for home (in minutes since midnight)"
8.      input intNightOut

9. return
----------------------------------------------------------------------------------------------------
0. processHoursMins()

1.      intHours = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) \ 60
2.      intMins = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) MOD 60

3. return
----------------------------------------------------------------------------------------------------
0. outputHoursMins()

1.      output "The number of hours worked is " + intHours
2.      output "The number of minutes worked is " + intMins

3. return
----------------------------------------------------------------------------------------------------
```

Flowchart


START
Declarations
num intMorningIn
num intNoonOut
num intNoonIn
num intNightOut
num intHours
num intMins
inputData()
processHoursMins()
outputHoursMins()
STOP


inputData()
output "Please enter what time you come into work (in minutes since midnight)"
input intMorningIn
output "Please enter what time you left for lunch (in minutes since midnight)"
input intNoonOut
output "Please enter what time you came back from lunch (in minutes since midnight)"
input intNoonIn
output "Please enter what time you left for home (in minutes since midnight)"
input intNightOut
return


processHoursMins()
intHours = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) \ 60
intMins = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) MOD 60
return


outputHoursMins()
output "The number of hours worked is " + intHours
output "The number of minutes worked is " + intMins
return

- As much as we are currently learning that declarations need to occur at the beginning of a program - this thought can and will change

- If you decide to declare variables, or constants within a module, then it must be noted that these variables are only visible inside the module.

- We call this the "**scope of a variable**" - and adding to the thought above, we will now also be introduced to the ideas of global and local variables
- a local variable is only accessible where it is declared
- a global variable would be accessible to all modules including the main program, declared in the main program in pseudocode (*and in real programming languages, we will note that the declaration is a little different*)

- names of global and local variables will also require much practice and learning to understand how to use them effectively and be able to differentiate between them. The textbook also states that programmers try to avoid global variables to minimize errors, but it does serve a purpose and needs to be understood.

- when the time comes to talk more about procedures and functions and their purpose in a program - we will also learn how this can lead to self-contained units of code that could effectively be used by other programs as required (*API & Class Library*)

- There is a general structure that is usually followed when developing modules for a basic program, the words input, processing, and output help a little

- The prescribed textbook offers the following explanation:
-- within the mainline logic (main program)

    declare global variables and constants.

    Housekeeping Tasks - steps to perform at the beginning of a program to get ready for the rest of the program.
    Detail loop Tasks - the core work of the program
    End-of-Job Tasks - steps to take at the end of the program to finish the application

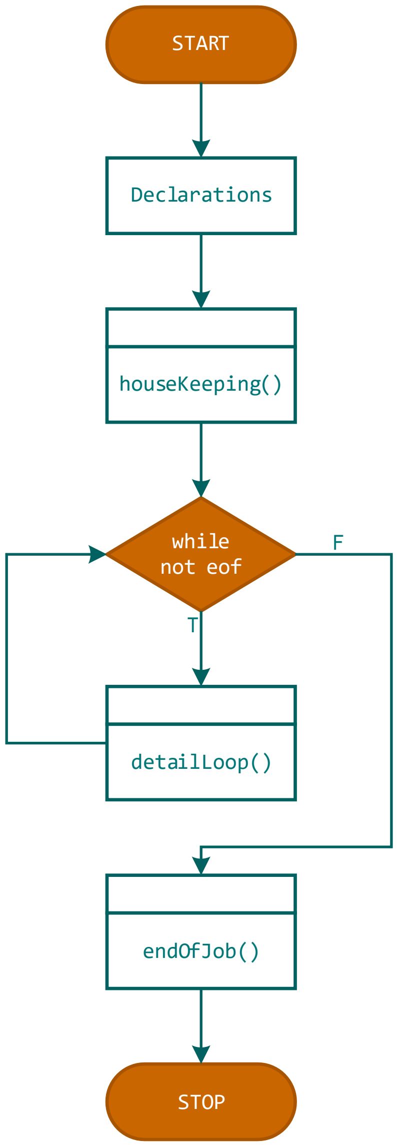
- Because of the above-mentioned idea of modules, the author already wishes that you understand the concept of a loop logic structure here. But we have yet to learn the basics first, develop and practice other programs before reaching this point. With this being said, please note the following code and explanations, but to also note that the full explanations will come at a later stage in our learning.

** Side note - the drawing of our flowcharts will be different compared to the textbook - but the logic and understanding will be the same

Example pseudocode

```
        0. start
        1. Declarations
        2.      houseKeeping()
        3.      while not eof
                        detailLoop()
                end while
        4.      endOfJob()
        5. stop
```

Example flowchart



Hierarchy charts
===

- This chart is a visual representation of how the modules are related to one another in a program

- Helps to see which modules exist in a program and to also note which modules also call other modules
- Remember in one of the statements made above, I mentioned that for now, the modules will be called from within the main program, but later in our learning, we will see that a module may also be called from within another module, so calling can happen wherever it is needed

- Each module should contain the tasks to accomplish one function, such as entering data, printing results or calculating results.

- There will always be one module that controls the flow to the other modules. The is the block placed at the top level (cantered) - usually known as the Control or Main module.

****
- Today, the solutions that are developed, are developed using a graphical user interface. And while in this type of solution, we would still use the idea of modules, they are not immediately related in the levels in the diagram we are expected to learn here. The user has more control of what to do next.

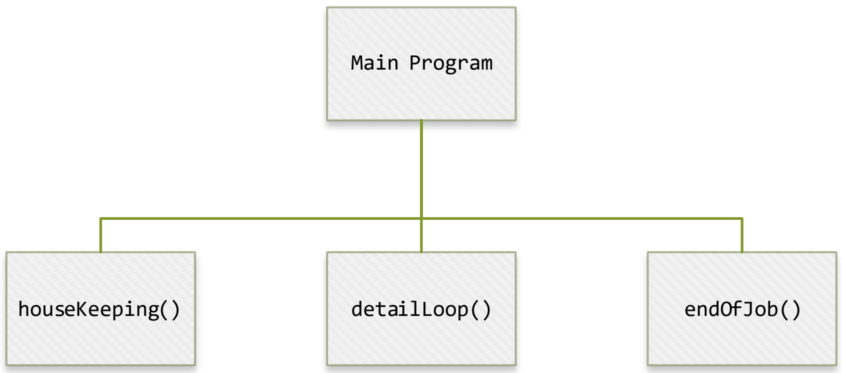
- Back in the day, solutions were procedural in nature - A top-down method was used, and the program was in more control - forcing us to follow a sequential order of execution.

- This is what we use to teach and get you to understand the flow of a program initially.

- Even with programming languages changing their paradigm from procedural to object-oriented programming, and users being in more control, modules still exist and plays it part in the development of a solution

****

- Divide the problem into sub-tasks (remember the words, input, processing, and output).
- You illustrate them in the order in which they will be processed.



Features of good program design
===

- The following should be noted as a checklist to assist with learning and developing good programs

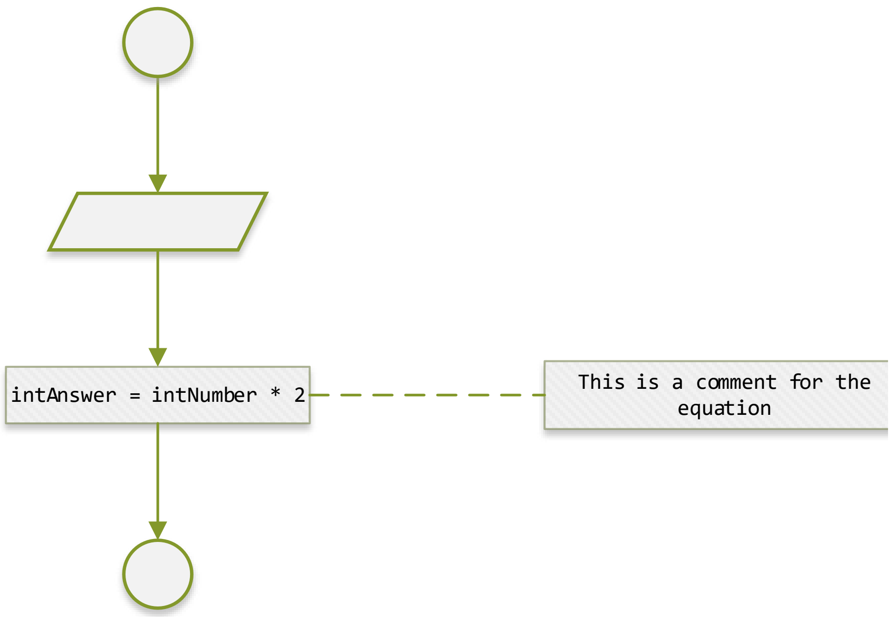
        - Use program comments where appropriate
                - comments help to document your program and serves the idea that when other programmers (including your lecturer at the moment) review your program, they have an understanding of the logic you have used in the solution chosen

                - Comments are not recognized by the compiler / interpreter

                - Syntax within comments differs amongst programming languages
                - Example

```
        // comment line

        num fltVAT = 0.15 // constant VAT value assigned

        /*
         * multi line
         * comment
        */
```

                - Flowchart (Annotated symbol)
                - ... we do have the ability to mention comments within a flowchart
                - dashed line connected to a 3-sided rectangle



        - Identifiers should be chosen carefully
                - choose a noun that represents the thing
                - Verbs are preferred for module names since they likely perform an action
                - by using meaningful, descriptive names, your program becomes self-documenting
                - use pronounceable names
                - use abbreviations wisely
                - avoid digits in a name
                - Follow a standard for naming - calcAnswer(), fltHourlyWage
                - With regards to constants - we will be using capital letters, but still using the prefix idea we are currently using in our lessons - fltVAT
                - when the time comes, a separate document, detailing an entire list of all the variables using a program will be developed and it is called a Data Dictionary

- Design clear statements within your program and modules
    - Avoid confusing line breaks
    -- for example - if we are meant to output an answer with quite a long sentence then consider the following

```
output "The number of hours worked is " + intHours +
       " and the number of minutes is " + intMins
```

instead of

```
output "The number of hours worked is " + intHours + " and the number
        of minutes is " + intMins
```

** Remember the hyphen being used during school days ... here in this output, we used the + (concatenation symbol) like the hyphen

- We could also use variables to store long values
-- for example, let's say we are meant to output the address for a company, based on how we are learning code at the moment

```
output "Address - School of Information Technology, Varsity College, 1 Benmore Drive, Sandton, 0000"
```

we could create a variable.

```
string strCompanyAddress

strCompanyAddress = "School of Information Technology, Varsity College, 1 Benmore Drive, Sandton, 0000"
```

and then have the following output which is a little more understandable.

```
output "Address: " + strCompanyAddress
```

- The textbook also offers an example, instead of long complicated equations, consider calculating a piece of the calculation before completing the remained of the calculation in another equation

- Write clear prompts and output statements
    - prompts are messages displayed on a monitor to ask the user for a response
    - These are used in command-line programs which we are currently learning and in the future in GUI (graphical user-interface) programs, but the prompts will differ slightly
    - The idea of echoing input is to repeat the input back to the user in an output and a subsequent prompt to further continue with the processing of a program

** Remember the program we completed in class - the same one shared earlier in this document

```
0. Start
1. Declarations
        num intMorningIn
        num intNoonOut
        num intNoonIn
        num intNightOut
        num intHours
        num intMins

2.      output "Please enter what time you come into work (in minutes since midnight)"
3.      input intMorningIn

4.      output "Please enter what time you left for lunch (in minutes since midnight)"
5.      input intNoonOut

6.      output "Please enter what time you came back from lunch (in minutes since midnight)"
7.      input intNoonIn

8.      output "Please enter what time you left for home (in minutes since midnight)"
9.      input intNightOut

10.     intHours = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) \ 60
11.     intMins = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) MOD 60

12.     output "The number of hours worked is " + intHours
13.     output "The number of minutes worked is " + intMins

14. Stop
```

- Imagine if we had left out the prompts and the statements in the output

```
0. Start
1. Declarations
        num intMorningIn
        num intNoonOut
        num intNoonIn
        num intNightOut
        num intHours
        num intMins

2.      input intMorningIn
3.      input intNoonOut
4.      input intNoonIn
5.      input intNightOut

6.      intHours = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) \ 60
7.      intMins = ((intNoonOut - intMorningIn) + (intNightOut - intNoonIn)) MOD 60

8.      output intHours
9.      output intMins

10. Stop
```

- while it feels like we saved on typing many lines, the execution of the program for the end-user will not make much sense, how will the user know what input is required if they are not asked for the value.
- and the same for the output, displaying the hours and minutes is okay, but will be user know what these displayed numbers represent

```
        - Maintain good programming habits as you develop your programming skills
                - Plan before you code
                -- Remember the Program Development Cycle
                -- and learn the 6 Problem Solving Steps
                1. Identify the Problem
                2. Understand the Problem
                3. Identify alternative ways to solve the problem
                4. Select the best way to solve the problem from the list of alternative solutions
                5. List the instructions that enable you to solve the problem using the selected solution
                6. Evaluate the solution
```

==

A little extra

==

We will learn that not all problems are the same and therefore their solutions are likely to differ as we develop programs.

Some problems are solved through a series of actions. These types of solutions are called algorithmic solutions. Once we have chosen the best method, then we have developed an algorithm.

Some problems require reasoning built on knowledge and experience and using a process of trial and error. This would mean that after the first time of coming up with a solution, we will evaluate our solution and using what we have learnt, change some processes to now have a better solution than the previous version and to possibly cater to new needs, if required. So, solutions cannot be reached through a direct series of steps, and this is called, heuristic solutions.

We can use both type of solutions in Step 6 - making sure we continuously develop better solutions.

==

- Maintain the habit of writing the pseudocode first and then the flowchart - helping you to visualize the flow of your program you have developed

- Desk-check your program logic on paper (we will be learning this in the future - will call this trace-tables)

- Think about the names of variables and modules as you develop a solution
- Design your program statements that will hopefully be easy to read and use