Chapter 5 Recap
===============

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

===
Overview
===
Chapter highlights:
- Introduction to Loop Logic
- Common Tasks Executed Using Loops
-- Incrementing (& decrementing)
-- Accumulating
-- Sentinel Value / Flag / Indicator
- while..loop
- do..while loop
- for..loop
- Using a loop control variable
- Nested loops
- The advantages of looping
- The logic of a loop
- Avoiding common loop mistakes
- Common loop applications
- The similarities and differences between selections and loops

===
* Please note that the following recap and associated notes are compiled from multiple sources. Also, that some of the work mentioned in the following sections has been discussed in the previous chapter notes including the homework that you have been completing. Do your best to compare and relate the notes so that you have a much clearer picture of the content shared.
===

===
Introduction to Loop Logic
===
-- Most problems involve doing this same thing over and over again, likely using a different set of data/input each time.
-- Companies can process reports, payrolls, mailing lists, stock takes, etc. using the loop logic structure.
-- While it is important to learn that we have the loop logic structure available for us to use, it is also now important to identify which instructions should be repeated and also in what order (sequential logic & decision logic is built upon now with the inclusion of loops).

-- There are 3 types of loop logic structures:
        - the while..loop, repeats instructions while a condition is true and stops repeating when the condition results in being false,
        - the do..while loop, similar to the logic of the while, repeats the instructions when the result of the condition is true and stops repeating when the condition is false, except there is one important difference, in that this do.while loop will execute the instructions inside it at least once before testing the condition to decide to repeat or not, whereas the while tests the condition before allowing entry to the loop.
        - The third type of loop is the for..loop, in which a variable is set to a given number and will repeat with the given incremental value to an ending number. When the variable is greater than the ending number, the loop ends.

-- Each of the 3 types of loops has a specific use to the logic of the problem we are faced with moving forward.

Common Tasks Executed Using Loops

-- There are several types of tasks that are executed through the use of the loop structures.

- Incrementing & decrementing - counting
- Accumulating - calculating a sum or total

-- In both tasks, a number is added or subtracted from a variable and the result is stored back into the same variable.

-- The difference between the 2 is that when counting, the value is a constant number & when accumulating the value is a variable, whose value is likely different each time it is repeated.

-- In each case, the resultant variable is initialized to zero before starting the loop, so that there is a starting point. (While this is true most of the time, there are times we might choose to initialize these variables to something other than 0. There would be logic that would make this make sense when the need arises)

Incrementing (& decrementing)

-- Incrementing or counting is done by adding a constant (usually the value is 1 and can be 2, if needed, can be something else)

```
        incrementing
                - (code) intCounter = intCounter + 1     (counting up)
        decrementing
                - (code) intCounter = intCounter - 1     (counting down)
```

-- Take note that the constant is the value, 1

-- The assignment statement (equation) allows the variable on the left to be assigned the result of the expression on the righthand side of the equal to (=) symbol

-- By using the above instruction, we can count any number of items (depending on the story). And because there is a variable that is accepting the result of the expression, each time the instruction is repeated, a new value is saved into the variable that is stated on the left.

Accumulating

-- Accumulating is the opportunity to sum up a group of numbers. Instead of using a constant value, the code, similar to incrementing, uses a variable value instead.

```
                - (code) intSum = intSum + intValue
                - (code) intTotalSales = intTotalSales + intSales
```

-- Take note that the variable is intValue and intSales

-- The expression adds the value of the variable to the sum variable and saved once again back into the accumulating variable.

-- It is important to note that the value in the variable is likely to be different each time the instruction is repeated (as compared to the incrementing note above where it always increments by a constant value).

-- Another idea related to accumulating is the product of a series of numbers.

```
                - (code) - intProduct = intProduct * intNumber
```

-- The above concept is used little - notes in other textbooks have spoken about the ideas around a topic of recursion, where accumulating using the product (*) will come in handy when it needs to.

==
Sentinel Value / Flag / Indicator
==
-- A value that represents an entry or exit point

-- The terms listed above, sentinel value, flag, indicator is all similar in nature. They are used to determine the course of action a program takes when this instruction is executed.

-- When we speak of loops, it is important to recognize that the loops are controlled by the value of a variable.

-- For example, we have seen in previous notes/programs, where we have read about allowing a user to continuously enter data and when a sentinel value has been entered, you need to stop processing. So we look at the sentinel value as being a value that represents just this, the ability to stop processing a section of code.

        - Example
```
        0. Start
        1. Declarations
                num intNumber
                num intSum

        2.      output "Please enter a number (999 to stop)"
        3.      input intNumber

        4.      intSum = 0
        5.      while intNumber <> 999
                        intSum = intSum + intNumber
                        ...

        6.      ...
```

** Fair enough, the discussions about the loop code is still going to be detailed below, but what you see here is an opportunity to read in a set of numbers, using the variable intNumber to process the input. The user is given a choice, that when they have completed entering the set of numbers, they are required to enter the value, 999, to stop the processing of the loop. The condition, which you can see in instruction #5, if the number is equal to 999, then the condition will result in being false and the loop will stop processing instructions inside of it and proceed to the next instruction outside the loop.

-- As mentioned above, the other terms, flag and indicator, play a similar role in a different storyline.

-- **Flag** - For example, imagine being asked to guess a number that was told to us, programmers, but not the user. The user would have to continuously guess until they guess the right number. Unlike in the previous example where the user is entering a set of numbers, here the user must keep on entering until they guess correctly. So now the idea of a sentinel value being in control is not going to work. But the idea of a flag will help.

- Example (**pseudocode)

```
0. Start
1. Declarations
        num intNumber
        num intGuess
        num intFlag

2.      intNumber = 143         // the number the user needs to guess
3.      intFlag = 0             // value set - 0 represents not guessed

4.      output "Please enter your guess"
5.      input intGuess

6.      while intFlag = 0       // allowed to enter because the number has not been guessed
                if intGuess = intNumber then
                        intFlag = 1     // if guessed, change the value of the flag
                        output "You guessed the right number"
                else
                        output "Guess not correct"
                        output "Please enter your guess (again) "
                        input intGuess
                endif
        endwhile

7.      output "End program"
8. Stop
```

** This time around the flag controls the need to repeat the instructions inside the loop. The user is not in control, like in the previous example. The value of the flag was solely my choice, the programmer. The flag variable was initialized to 0 to represent that the value being guessed has not been found and note in the if..then statement inside the loop, the flag will change to 1 if the guess is correct.

****The opportunity to use such a concept will be your choice, the programmer.**
What is the purpose of the flag?
Where is it going to be used?
What do the values represent?

-- The word indicator could be used as a logical variable that a programmer sets within a program to change the processing path or to control the processing of a loop. The user has no knowledge of these indicators during processing. They are used to detect changes in the output, in the processing of data and so on.

-- One of the possible uses of an indicator, is to have an error indicator, which can used to inform the program that an error has occurred in the input or the output during processing of a program.

-- Another - If we are reading a file, we could utilize an indicator to inform the program that there is no more data to be read.

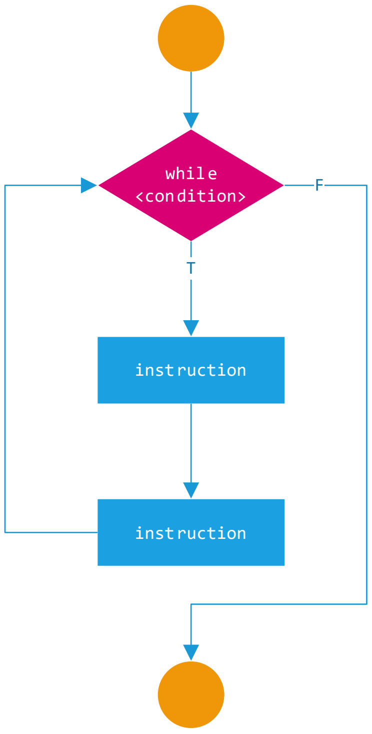
-- Other terms that we might come across in the future, switches, and trip values

-- As a programmer, you need to understand the need to develop a program that fits the needs of the solution and whatever it is you need to implement, you will need to implement as required.

==
while..loop
==
-- The first of the 3 types of loops. This type of loop tells the computer that while the condition is 'true', repeat all the instructions inside the while..loop structure. When the result of the condition is false, the program will 'exit' the loop and proceed to execute the next available instruction.

```
        ** basic structure
        while <condition>
                instruction
                instruction
                ...
                ..
        endwhile
```

```
        ** flowchart
```



```
        ** Note - use indentation to improve the readability of the algorithm
```

-- Another term that we come across when speaking of the while..loop is that it is a "pre-test" loop. This means that the condition is evaluated first before allowing the program to enter the loop. So, it is entirely possible that a program may also never enter and execute the instructions in the loop because the result of the condition was false to begin with. You can see this in the design of the flowchart for the while..loop.

-- We use the while..loop when we do not know the number of times the instructions will be repeated (example using the sentinel value), or there are cases when the instructions in the loop should not be processed (example using the flag).

-- Once again, the order of instructions within the loop is just as important when we learnt the sequential logic structure (at the time when we did not know of these other structures). If the instructions within the loop are not executed in the order that we have researched, we could end up with a logical error or even a run-time error.

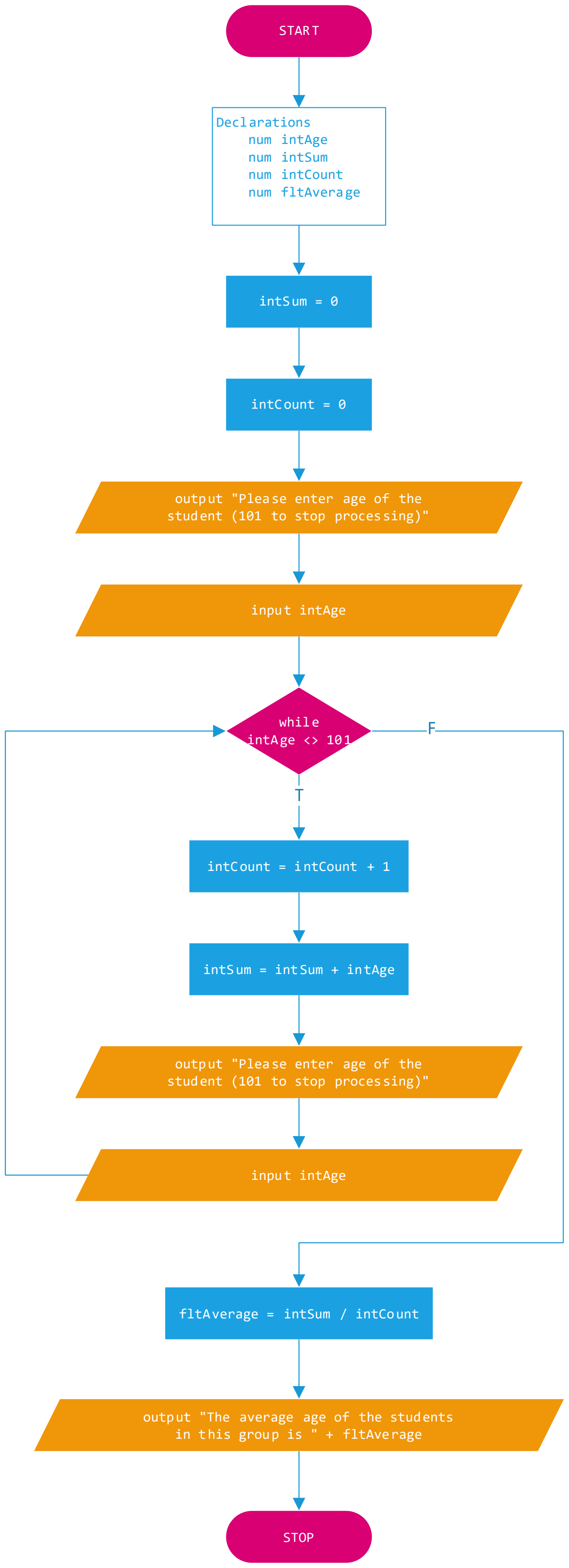
***** using a sentinel value***

- The following example is to determine the average age of all the students in a group. Use the age input of 101 to stop the processing when you have entered all the ages.

**pseudocode

```
0. Start
1. Declarations
        num intAge
        num intSum
        num intCount
        num fltAverage

2.      intSum = 0      // initialized with a starting number
3.      intCount = 0    // initialized with a starting number

4.      output "Please enter age of the student (101 to stop processing)"
5.      input intAge

6.      while intAge <> 101
                intCount = intCount + 1  // incrementing - counting the number of students
                intSum = intSum + intAge // accumulating - the ages of all the students
                output "Please enter age of the student (101 to stop processing)"
                input intAge
        endwhile

7.      fltAverage = intSum / intCount
8.      output "The average age of the students in this group is " + fltAverage

9. Stop
```

**flowchart


START
Declarations
num intAge
num intSum
num intCount
num fltAverage
intSum = 0
intCount = 0
output "Please enter age of the student (101 to stop processing)"
input intAge
while intAge <> 101
F
T
intCount = intCount + 1
intSum = intSum + intAge
output "Please enter age of the student (101 to stop processing)"
input intAge
fltAverage = intSum / intCount
output "The average age of the students in this group is " + fltAverage
STOP

- 101 is the sentinel value.

- The following 2 instructions listed below, from the program, are interesting, this could be called the **primer read**, because it is what gives the while..loop a value for the variable in order for the condition to be true the first time to enter the loop.

```
4.      output "Please enter age of the student (101 to stop processing)"
5.      input intAge
```

- Once inside a loop, the program will never leave until the result of the condition is false.
- Because of this above statement, can you see why now the prompt (output) and inputting of the age is repeated inside the loop.

- Take note of the order of instructions before entering the loop and also once inside the loop.

- Have look at the order of instructions in the loop below. We attempt to do the same thing, but can you see why the following order of instructions inside the loop would be incorrect

```
5.      ...
6.      while intAge <> 101
                output "Please enter age of the student (101 to stop processing)"
                input intAge
                intCount = intCount + 1  // incrementing - counting the number of students
                intSum = intSum + intAge // accumulating - the ages of all the students
        endwhile
7.      ...
```

- What type of error would you expect to see if we go ahead with the above code? (logical!)
- Something to genuinely think about

-- Remember that when we choose to use a sentinel value, the loop becomes an indefinite loop that will execute a different number of times each time the program executes

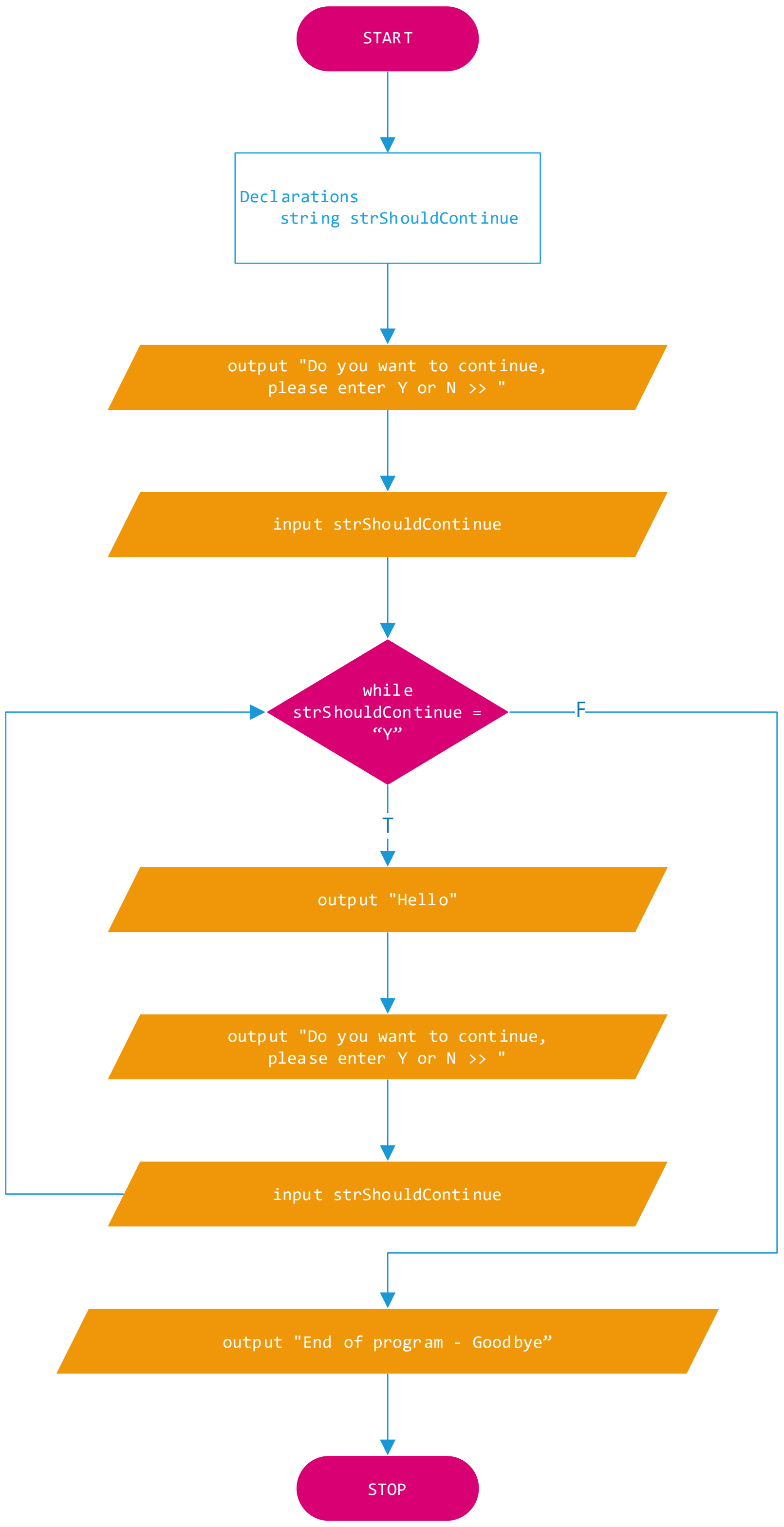
-- Most of the time, it is the user that will decide how many times to repeat the instructions within the loop *(but once again, there are opportunities for a loop to be definite and not be controlled by the user)*

* *Another example*
- The following program requests from the user, an input of 'Y' (yes) or 'N' (no) and the loop is dependent on this input, effectively making 'N' (no) the sentinel value that would end of processing of the loop.

** pseudocode

```
0. Start
1. Declarations
        string strShouldContinue

2.      output "Do you want to continue, please enter Y or N >> "
3.      input strShouldContinue // primer read to initiate entrance to the loop

4.      while strShouldContinue = "Y"
                output "Hello"
                output "Do you want to continue, please enter Y or N >> "
                input strShouldContinue // repeated to decide to continue repeating or not
        endwhile

5.      output "End of program - Goodbye"

6. Stop
```

** flowchart


START
Declarations
string strShouldContinue
output "Do you want to continue, please enter Y or N >> "
input strShouldContinue
while strShouldContinue = “Y”
F
T
output "Hello"
output "Do you want to continue, please enter Y or N >> "
input strShouldContinue
output "End of program - Goodbye”
STOP

- The above example is like the one before, the loop is dependent on the user's input.

**** using a counter**

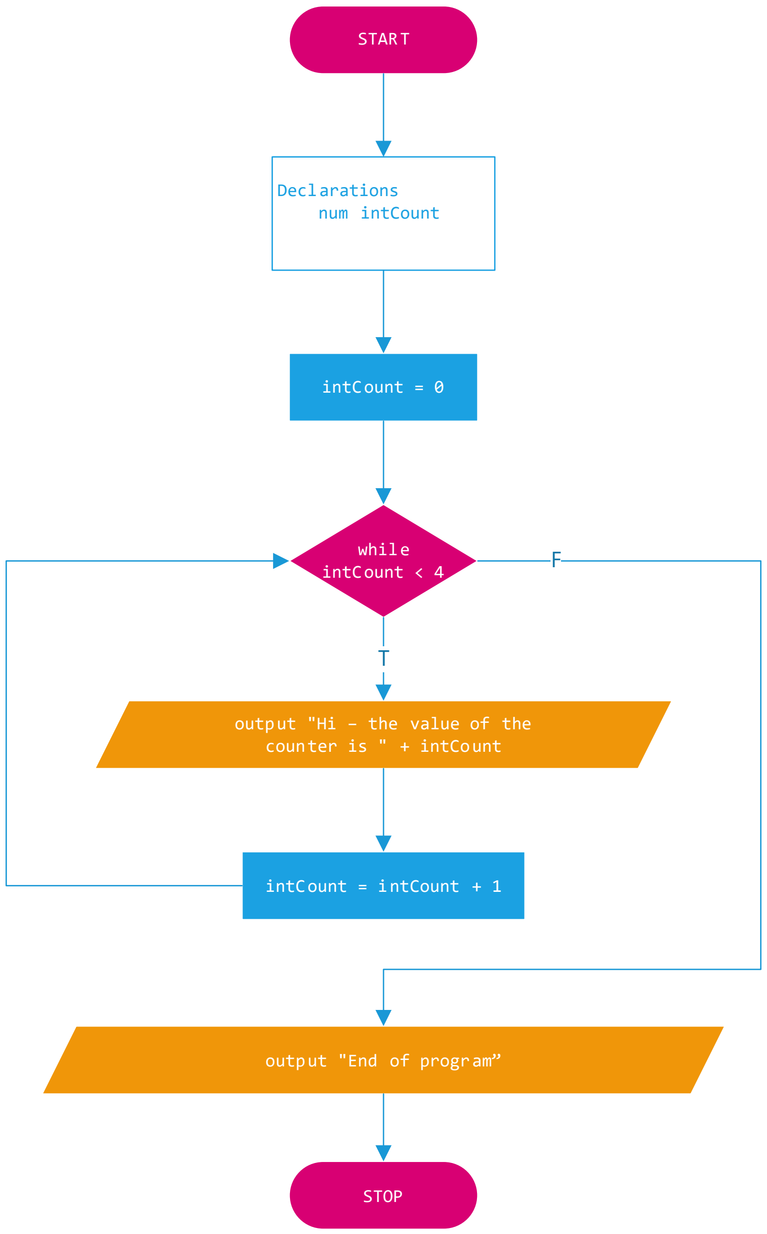
- Now while we understand that a while..loop is likely to be used when we do not know how many times the instructions are to be repeated, it is possible to change that perception.

- We are able to use the while..loop to create a definite loop that will execute a predetermined number of times.
- This will mean that the condition will use a counter variable to control the repetitions
- We will increment the counter - counting up, for example, from 1 to 10 ... but there are times when we might be required to count down, for example, from 10 t0 1, if the need to count backwards is expected.
- *Counter variables need to be initialized to 0 before we begin counting.*

** pseudocode

```
0. Start
1. Declarations
        num intCount

2.      intCount = 0    // initialized since it is going to be used to control the loop

3.      while intCount < 4
                output "Hi - the value of the counter is " + intCount
                intCount = intCount + 1 // incrementing the counter / control variable
        endwhile

4.      output "End of program"

5. Stop
```

** flowchart



* The for loop that is going to be discussed later, is the better choice when we know the number of times to repeat instructions, but this would be an interesting choice for you as a programmer to make.

***** A possible logical error***

- We are taking the same example of calculating and outputting the average age of the students in a group. The loop like before is dependent on the intAge variable that is accepting input from the user.

- We do assume that the user is going to enter the students ages as expected and when they are done with the input, then enter the sentinel value of 101 to end the processing of the loop.

- However, what if the user enters the sentinel value at the beginning of the execution of the program.
- This would mean that we never enter the loop and the calculation of fltAverage will be an expression that is going to be 0 / 0 - anything being divided by zero will result in an error or possibly an undefined result.

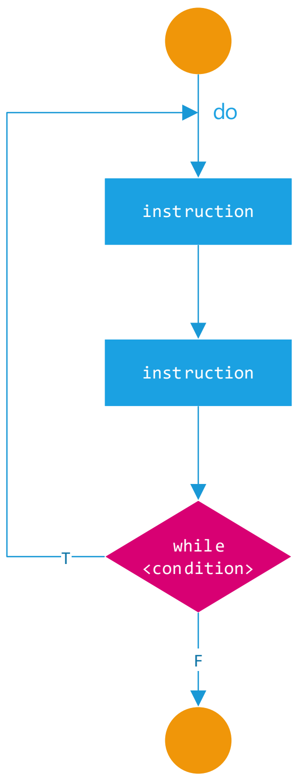
- That would not be the best output for the user, so take note of the addition of the if..then statement after the loop, the condition compares the count variable to zero. If it is equal to zero, then it means nothing was entered (our logic, our understanding), the incrementing instruction was never executed and therefore means that there is no average to display, but if the loop was entered into, the count would have incremented meaning that a valid age was entered and there is an average that will be displayed.

**pseudocode

```
0. Start
1. Declarations
        num intAge
        num intSum
        num intCount
        num fltAverage

2.      intSum = 0      // initialized with a starting number
3.      intCount = 0    // initialized with a starting number

4.      output "Please enter age of the student (101 to stop processing)"
5.      input intAge

6.      while intAge <> 101
                intCount = intCount + 1  // incrementing - counting the number of students
                intSum = intSum + intAge // accumulating - the ages of all the students
                output "Please enter age of the student (101 to stop processing)"
                input intAge
        endwhile

7.      if intCount <> 0 then
                fltAverage = intSum / intCount
                output "The average age of the students in this group is " + fltAverage
        else
                output "No data was entered. There is no average to display"
        endif
8. Stop
```

==
do..while loop
==
-- The next loop is the do..while loop, similar in logic to the while.loop, with one notable difference, the condition is evaluated at the end of the loop.
-- If the result of the condition is true, the instructions inside the loop is repeated and if the result is false, the loop is exited and the next instruction is executed.

```
        ** basic structure
        do
                instruction
                instruction
                ...
                ..
        while <condition>

        ** flowchart
```



-- Another term we associate with the do..while loop is that it is a post-test loop, evaluating the condition after the instructions in the loop is executed at least once.

-- It is important to distinguish between the two loops mentioned above, so that you can use them appropriately.

-- Take note of the differences - when you use the while..loop, you must initialize the data so that the condition results in being true in order to enter the loop and execute the instructions inside of it. - when you use the do..while loop, you may choose to deal with data in a different way you see fit, since the condition is, again, tested at the end of the loop, after executing the instructions inside of it at least once.

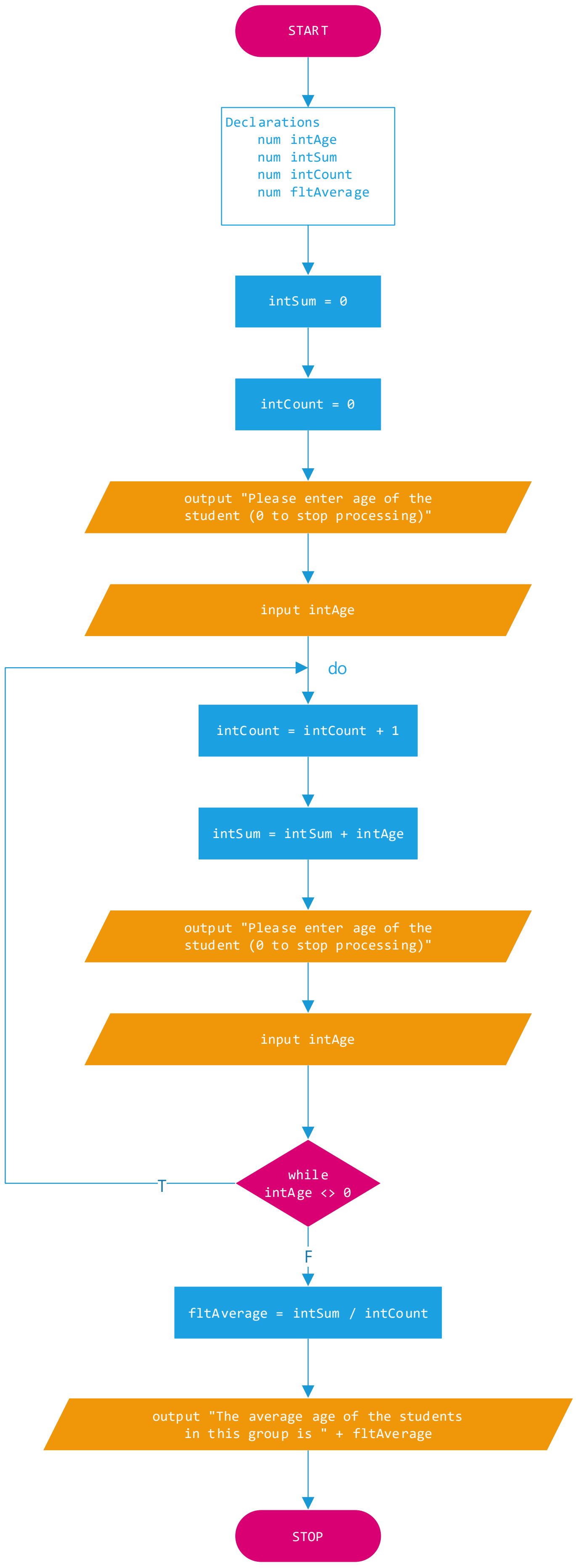
** With the while..loop, we do not really know how many times the loop will be executed, but we also don't know if the code inside will be executed at all, whereas with the do..while loop, the code inside the loop will be executed at least once.

- The following example is again, the calculation and output of the average age of students in a group, using the do..while loop

**pseudocode

```
0. Start
1. Declarations
        num intAge
        num intSum
        num intCount
        num fltAverage

2.      intSum = 0        // initialized with a starting number
3.      intCount = 0      // initialized with a starting number

4.      output "Please enter age of the student (0 to stop processing)"
5.      input intAge

6.      do
                intCount = intCount + 1  // incrementing - counting the number of students
                intSum = intSum + intAge // accumulating - the ages of all the students
                output "Please enter age of the student (0 to stop processing)"
                input intAge
        while intAge <> 0

7.      fltAverage = intSum / intCount
8.      output "The average age of the students in this group is " + fltAverage

8. Stop
```

** flowchart


START
Declarations
num intAge
num intSum
num intCount
num fltAverage
intSum = 0
intCount = 0
output "Please enter age of the student (0 to stop processing)"
input intAge
do
intCount = intCount + 1
intSum = intSum + intAge
output "Please enter age of the student (0 to stop processing)"
input intAge
while intAge <> 0
T
F
fltAverage = intSum / intCount
output "The average age of the students in this group is " + fltAverage
STOP

- Take note of the above code, not much has changed, except that the loop is different & the sentinel value was changed (**Was this a good thing?**).

- *We might think of the following*: Since the loop is only evaluating the condition at the end of the loop, is it necessary to ask the user for the 1st age before entering the loop. (The initializing of the incrementing and accumulating variables still need to executed at the beginning before the loop begins)

- *Take note of the following edited program below (& the difference to the above code), the input of the age is happening inside the loop at the beginning. Since we are learning that this loop will execute the code inside the loop at least once, would it be okay, to make this change in instructions.*

**pseudocode

```
0. Start
1. Declarations
        num intAge
        num intSum
        num intCount
        num fltAverage

2.      intSum = 0       // initialized with a starting number
3.      intCount = 0     // initialized with a starting number


6.      do
                output "Please enter age of the student (101 to stop processing)"
                input intAge

                intCount = intCount + 1  // incrementing - counting the number of students
                intSum = intSum + intAge // accumulating - the ages of all the students
        while intAge <> 101

7.      fltAverage = intSum / intCount
8.      output "The average age of the students in this group is " + fltAverage

8. Stop
```

- All looks to be correct, but if we attempt to execute the code, using all possible variations for the inputting of the age, there is small problem that could arise.
- Take note of the sentinel value (101) being used again, what if the user enters the following values for the age:

```
        1, 2, 3, & 101
        The loop will accept the
                - 1, the count will increment to 1 & the sum will accumulate to 1
                - 2, the count will increment to 2 & the sum will accumulate to 3
                - 3, the count will increment to 3 & the sum will accumulate to 6
                - 101, the count will increment to 4 & the sum will be 107 ...
                - the condition will be tested and exit the loop ...
```

- A logical mistake, we are never meant to count 101 or add it to the accumulating variable. This will result in the average producing & outputting the wrong answer.

- Another error, what if the initial input is 101

```
        The loop will accept the
                - 101, the count will increment to 1 & the sum will accumulate to 101
                - the condition will once again be tested and the loop will exit
```

- Another logical mistake, since the average calculation will take the sum 101 and divide it by 1, producing and outputting a value of 101 for the average which we know is incorrect.

- Realizing that the sentinel value might be a problem, could we change it to something else.

-- You would have noticed when the code for the do..while loop was shared, I had changed the sentinel value to 0. Could this be a solution (with the edited input of the age being inside the loop)?

**pseudocode

```
0. Start
1. Declarations
        num intAge
        num intSum
        num intCount
        num fltAverage

2.      intSum = 0        // initialized with a starting number
3.      intCount = 0      // initialized with a starting number


6.      do
                output "Please enter age of the student (0 to stop processing)"
                input intAge

                intCount = intCount + 1  // incrementing - counting the number of students
                intSum = intSum + intAge // accumulating - the ages of all the students
        while intAge <> 0

7.      fltAverage = intSum / intCount
8.      output "The average age of the students in this group is " + fltAverage

8. Stop
```

- If we enter the sentinel value to begin with ...
    - intAge accepts the 0,
    - the intCount increments to 1
    - the intSum accumulates to 0 &
    - the condition is tested, result is false, loop exits

    - the average calculates - 0/1 which equals to 0, which is now correct

- So changing the sentinel value to something more acceptable and not producing a logical mistake is the way to go

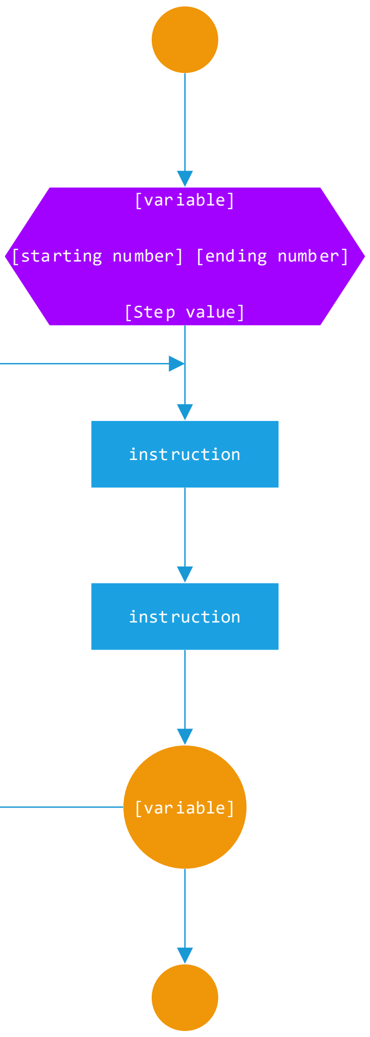
- What about including the if..then statement after the loop again to decide if an average is to be printed or not?

BUT, it is once again important to note that this is not going to always work. There will be times when the sentinel value cannot be zero and we would have to find other solutions to our logical order of instructions and the correct instructions to include into a program.

==
for..loop
==
-- This type of loop increments or decrements a variable each time the loop is repeated. Another name associated with the for..loop is the automatic-counter loop. A definite loop where we as the programmers know how many times the loop will be executed.

```
        ** basic structure
        for [variable] = [starting number] to [ending number] Step [Step value]
                instruction
                instruction
                ...
                ..
        endfor
```

```
        ** flowchart
```



-- A little simpler and better than using a while..loop to do the same job.
-- Compare the following code

```
        2.      intCount = 0
        3.      while intCount <= 3
                        output "Hello"
                        intCount = intCount + 1
                endwhile
```

and

```
        2.      for intCount = 0 to 3 Step 1
                        output "Hello"
                endfor
```

-- The for..loop provides multiple actions in one structure, namely:
- **initializing the variable with a starting value**
- **evaluates the variable (condition) to decide if it continues or exits the loop**
- **increments the counting variable using the Step information**

-- The Step value is the value by which the loop control variable changes - the default is always (+)1 but if the programmer wishes to increment the loop by another number at each iteration, then they need to specify the Step value as required. Please note that even though it is understood that the default is +1, it is important to state "Step 1" in your source code in the development of a program.

- Another simple example of a loop *'going backwards'* (there will come a time when we need to do just this)

```
        2.      for intCount = 10 to 1 Step -1
                        output intCount
                endif
```

-- Rules:
- When the computer executes the loop, it sets the counter variable equal to the beginning number
- When the computer reaches the 'endfor' statement, it increments the counter variable. The variable is incremented before the condition is processed (the condition that is built within the loop to decide to continue processing or end the execution of the loop).
- When the counter variable is less than or equal to the ending number, the processing continues.
- When the counter variable is greater than the ending number, the loop has ended and the next available instruction is executed.

** *side note*
- When the counter variable is being decremented, it is done at the time when the processing goes to the 'endfor' statement. When the counter variable is less than or equal to the ending number then the loop ends.

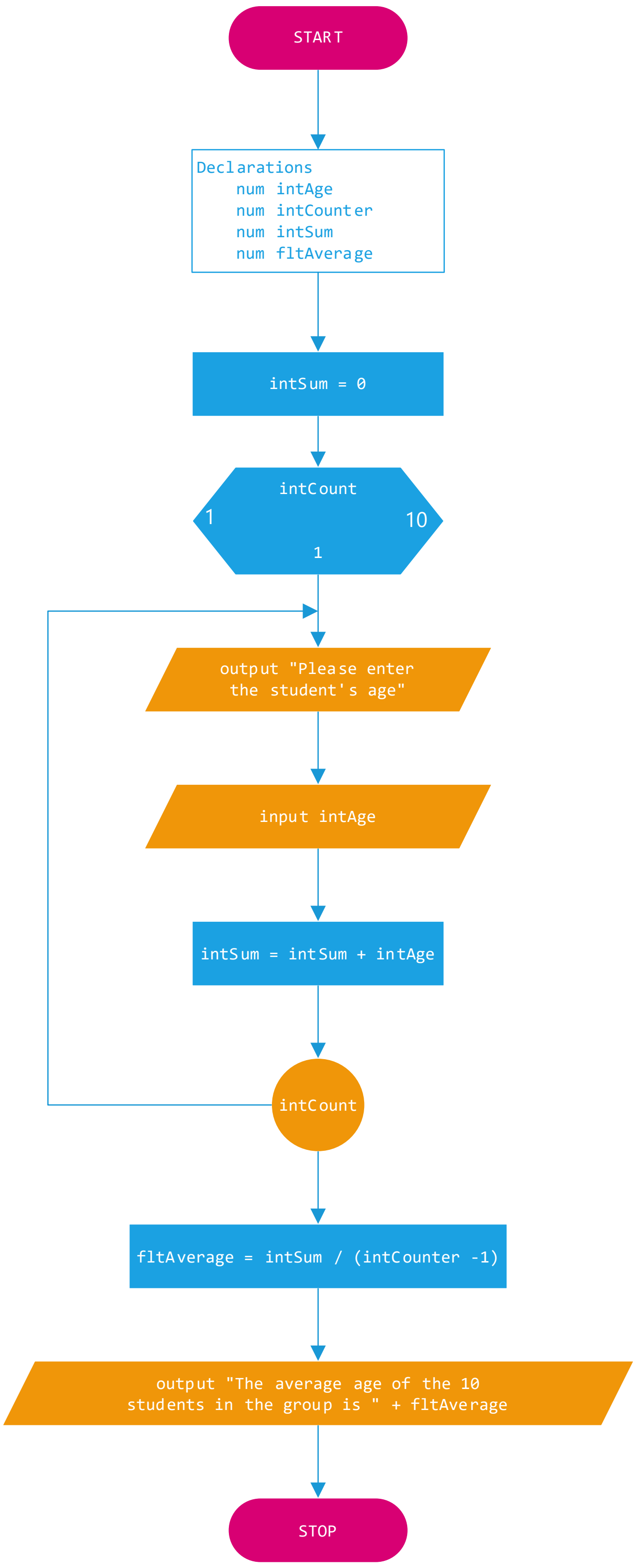
-- Again, when you know the number of times the loop will be executed, then the for..loop is the better choice.

- Consider again the example about the average age of the students in a group. If we know the number of students, then;

**pseudocode

```
0. Start
1. Declarations
        num intAge
        num intCounter
        num intSum
        num fltAverage

2.      intSum = 0

3.      for intCount = 1 to 10 Step 1
                output "Please enter the student's age"
                input intAge
                intSum = intSum + intAge
        endfor

4.      fltAverage = intSum / (intCounter - 1) // or intSum / 10
5.      output "The average age of the 10 students in the group is " + fltAverage

6. Stop
```

**flowchart


START
Declarations
num intAge
num intCounter
num intSum
num fltAverage
intSum = 0
intCount
1
10
1
output "Please enter the student's age"
input intAge
intSum = intSum + intAge
intCount
fltAverage = intSum / (intCounter -1)
output "The average age of the 10 students in the group is " + fltAverage
STOP

- in the calculation of the average - while we know that it should be divided by 10, take note of the code (intCounter - 1) - remember that when the loop ends, the counter variable is greater than the ending number in order for the loop to stop processing. So, in the above example, the counter variable would have the value of 11, so that is why we need to subtract 1

-- There are many variations and possibilities once again with the for..loop and given that it is important to learn all the loop logic structures, it is just as important to understand the decision to use one loop over another. Ask yourself, if this is the best choice for the solution at the moment? Can it be more efficient, if I would like it to be?

===
Using a loop control variable
===
** A little repetition of notes, but good to go through it again, once you start to understand the examples that are being presented.

-- the loops are all dependent on the condition .
(*the while..loop and the do..while loop have visible conditions and the for..loop has a condition built into its logic*)

-- The condition is likely to use a variable that is assessed at each iteration to decide if the loop continues repeating or stops the repeating & exits the loop to the next available instruction.

-- This variable is required to be initialized before entering the loop so that the code runs as planned.

-- Within the body of the loop, the variable that is being used to control the loop must have the chance to alter its value and therefore determine if the loop repeats at the next iteration or stops executing.

-- As it has been explained above, the repetitions can be controlled by:

- a counter
    - when you know the number of times the loop will be repeated.
- a sentinel value
    - when you do not know the number of times the loop will be repeated.
- a flag
    - when you do not know the number of times, but something within the coding will change and it will be decided for us.

** Please note that the use of a counter is related to the design of the while..loop and the do..while loop*

** We are learning, that the for..loop is better suited if we know the number of times a loop is meant to be repeated & when we do not know the number of times, then the while..loop and the do..while loop is better suited, but that does not stop us from using the while..loop and the do..while loop to act similar and control the loop with a counter. Revisit the examples shared to understand some of the possibilities.*

==

Nested loops

==

-- Similar to the ideas shared about nested if..then statements, we now also have the ability to nest looping structures.

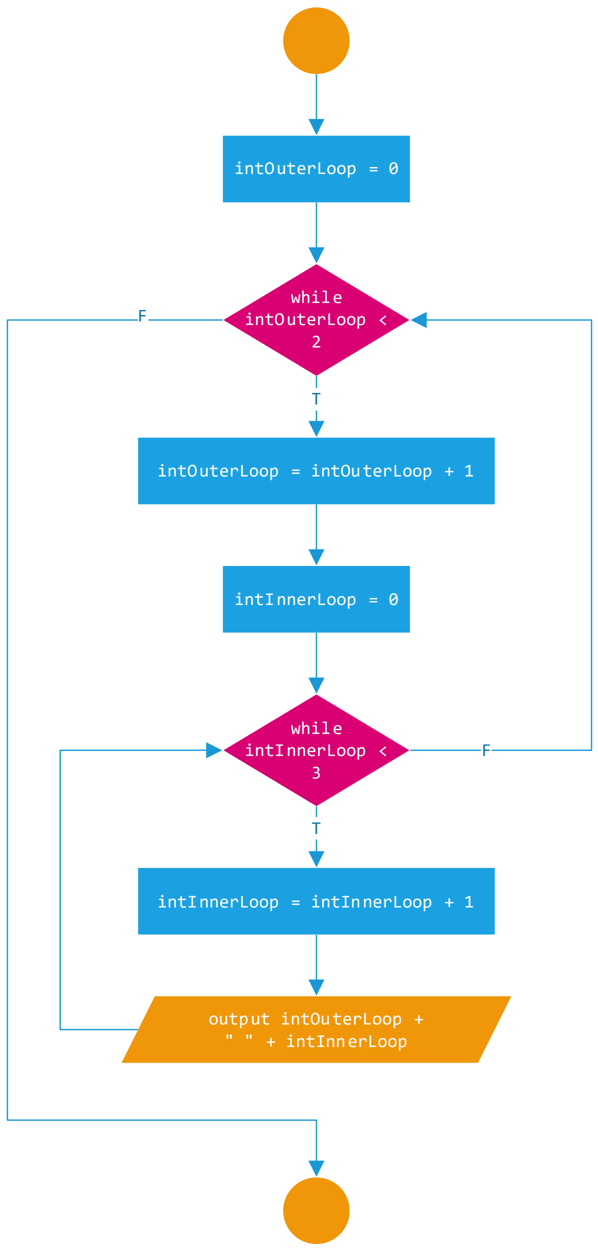
- Terms:
    - -- **Nested loops** - loops within loops
    - -- **Outer loop** - the loop that contains the other loop
    - -- **Inner loop** - the loop that is contained inside a loop
- a common use of nested loops is when we have 2 variables, and the repeat will produce every combination of values

-- The following are some basic variations and possibilities for nested loops

**pseudocode

```
        2.      intOuterLoop = 0

        3.      while intOuterLoop < 2
                        intOuterLoop = intOuterLoop + 1
                        intInnerLoop = 0
                        while intInnerLoop < 3
                                intInnerLoop = intInnerLoop + 1
                                output intOuterLoop + " " + intInnerLoop
                        endwhile
                endwhile
```

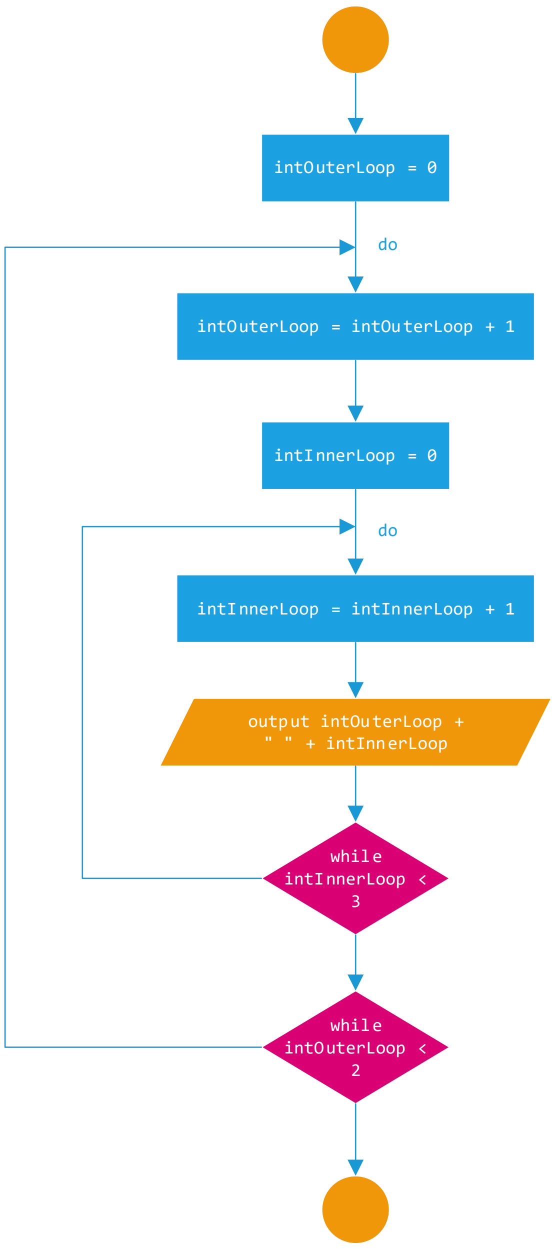
** flowchart


intOuterLoop = 0
while intOuterLoop < 2
F
T
intOuterLoop = intOuterLoop + 1
intInnerLoop = 0
while intInnerLoop < 3
F
T
intInnerLoop = intInnerLoop + 1
output intOuterLoop + " " + intInnerLoop

**pseudocode

```
2.      intOuterLoop = 0

3.      do
                intOuterLoop = intOuterLoop + 1
                intInnerLoop = 0
                do
                        intInnerLoop = intInnerLoop + 1
                        output intOuterLoop + " " + intInnerLoop
                while intInnerLoop < 3
         while intOuterLoop < 2
```

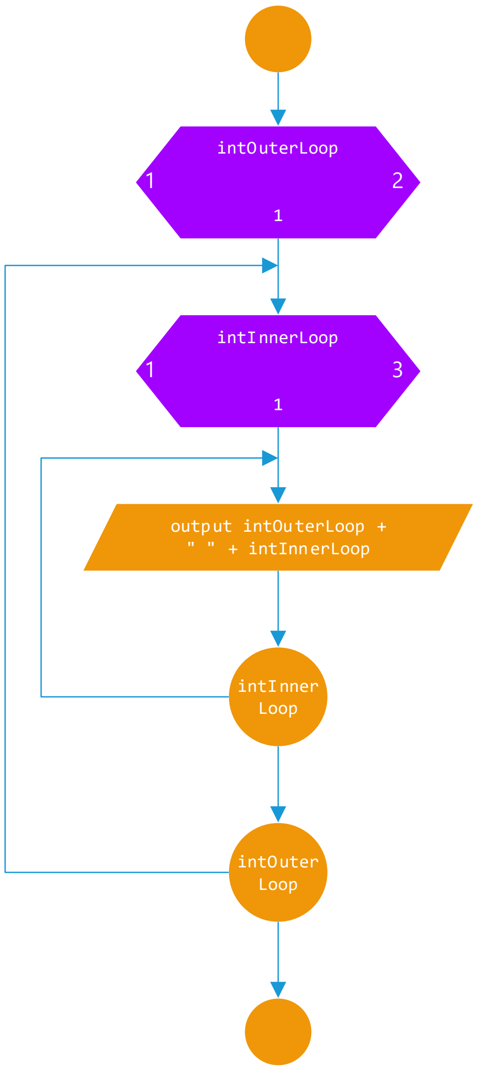
** flowchart


intOuterLoop = 0
do
intOuterLoop = intOuterLoop + 1
intInnerLoop = 0
do
intInnerLoop = intInnerLoop + 1
output intOuterLoop + " " + intInnerLoop
while intInnerLoop < 3
while intOuterLoop < 2

**pseudocode

```
2.      for intOuterLoop = 1 to 2 Step 1
                for intInnerLoop = 1 to 3 Step 1
                        output intOuterLoop + " " + intInnerLoop
                endfor
        endfor
```

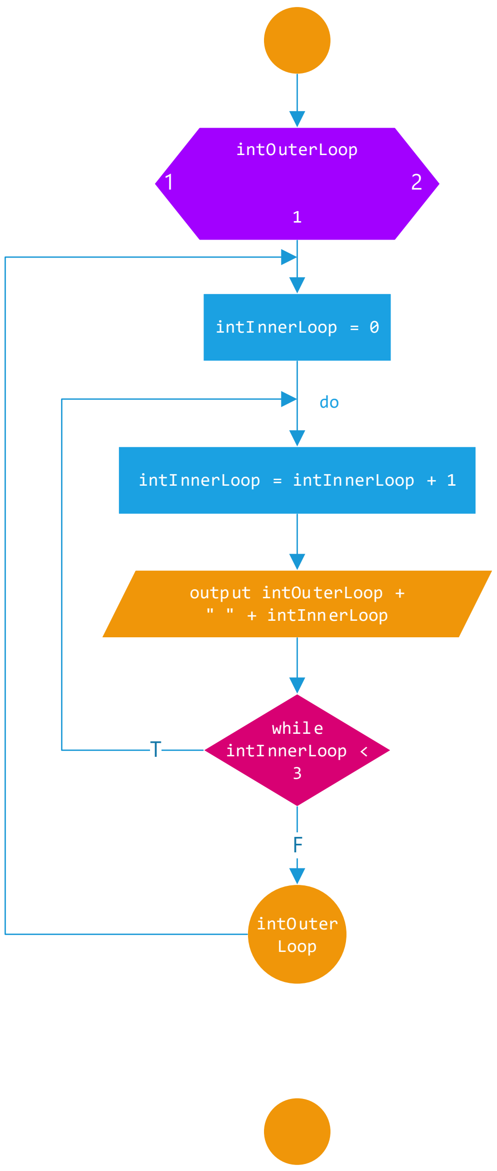
** flowchart



**pseudocode

```
2.      for intOuterLoop = 1 to 2 Step 1
                intInnerLoop = 0
                do
                        intInnerLoop = intInnerLoop + 1
                        output intOuterLoop + " " + intInnerLoop
                while intInnerLoop < 3
        endfor
```

**flowchart



intOuterLoop
1
2
1
intInnerLoop = 0
do
intInnerLoop = intInnerLoop + 1
output intOuterLoop + " " + intInnerLoop
while intInnerLoop < 3
T
F
intOuter Loop

- The example provided in the textbook is an interesting one: Imagine that there is a piece of homework to give out to students. There are different quizzes that students can complete, but all the quizzes follow the same structure, each quiz contains 5 main sections, and each section contains 3 questions.

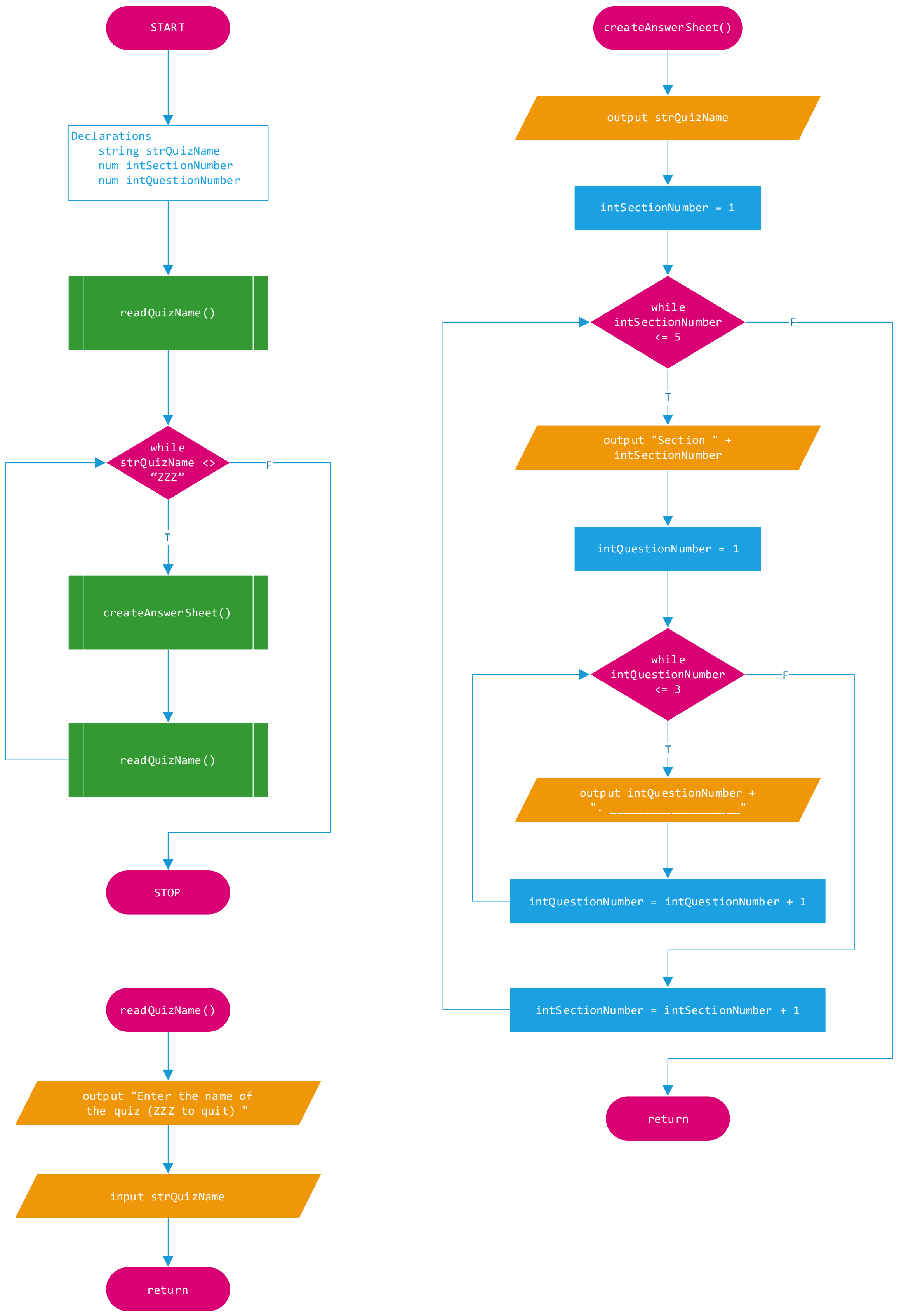
The user (a teacher) wishes to print out an answer sheet to hand out to the students. Each answer sheet must display the name of the quiz at the top and then a section heading and the question numbers (x3) and then print out the next section heading and question numbers, repeatedly for the 5 sections.

** pseudocode

*** the following program makes use of modules (some coding is changed as compared to the textbook - I won't be using the constants, and will be hardcoding some of the values in the coding of this version of the program - you may compare the work below to what is in the textbook)*

```
0. Start
1. Declarations
        string strQuizName
        num intSectionNumber
        num intQuestionNumber


2.      readQuizName()          // primer read

3.      while strQuizName <> "ZZZ"
                createAnswerSheet()      // to print each answer sheet as required
                readQuizName() // read for the next answer sheet or not (ZZZ)
        endwhile

4. Stop
----------------------------------------------------------------------------------------------------
0. readQuizName()
1.      output "Enter the name of the quiz (ZZZ to quit) "
2.      input strQuizName
3. return
----------------------------------------------------------------------------------------------------
0. createAnswerSheet()
1.      output strQuizName      // printing heading for specific answer sheet

2.      intSectionNumber = 1     // initializing the section to begin with section 1

3.      while intSectionNumber <= 5
                output "Section " + intSectionNumber     // print sub(section) heading
                intQuestionNumber = 1    // initialize question number to begin with 1

                while intQuestionNumber <= 3
                        output intQuestionNumber + ". ___________________" // space for ans
                        intQuestionNumber = intQuestionNumber + 1
                endwhile
                intSectionNumber = intSectionNumber + 1
        endwhile

4. return
----------------------------------------------------------------------------------------------------
```

** flowchart


START
Declarations
string strQuizName
num intSectionNumber
num intQuestionNumber
readQuizName()
while strQuizName <> “ZZZ”
T
F
createAnswerSheet()
readQuizName()
STOP
readQuizName()
output "Enter the name of the quiz (ZZZ to quit) "
input strQuizName
return
createAnswerSheet()
output strQuizName
intSectionNumber = 1
while intSectionNumber <= 5
T
F
output "Section " + intSectionNumber
intQuestionNumber = 1
while intQuestionNumber <= 3
T
F
output intQuestionNumber + ". ___________________"
intQuestionNumber = intQuestionNumber + 1
intSectionNumber = intSectionNumber + 1
return

```
--------------------------------------------------------------------------------------------------------
-- Considering that this above solution uses while..loops, noting that the loops that oversee the section
and question numbers (controlled by counter variables). The only thing that the user is required to do
is enter the name of the quiz.
```

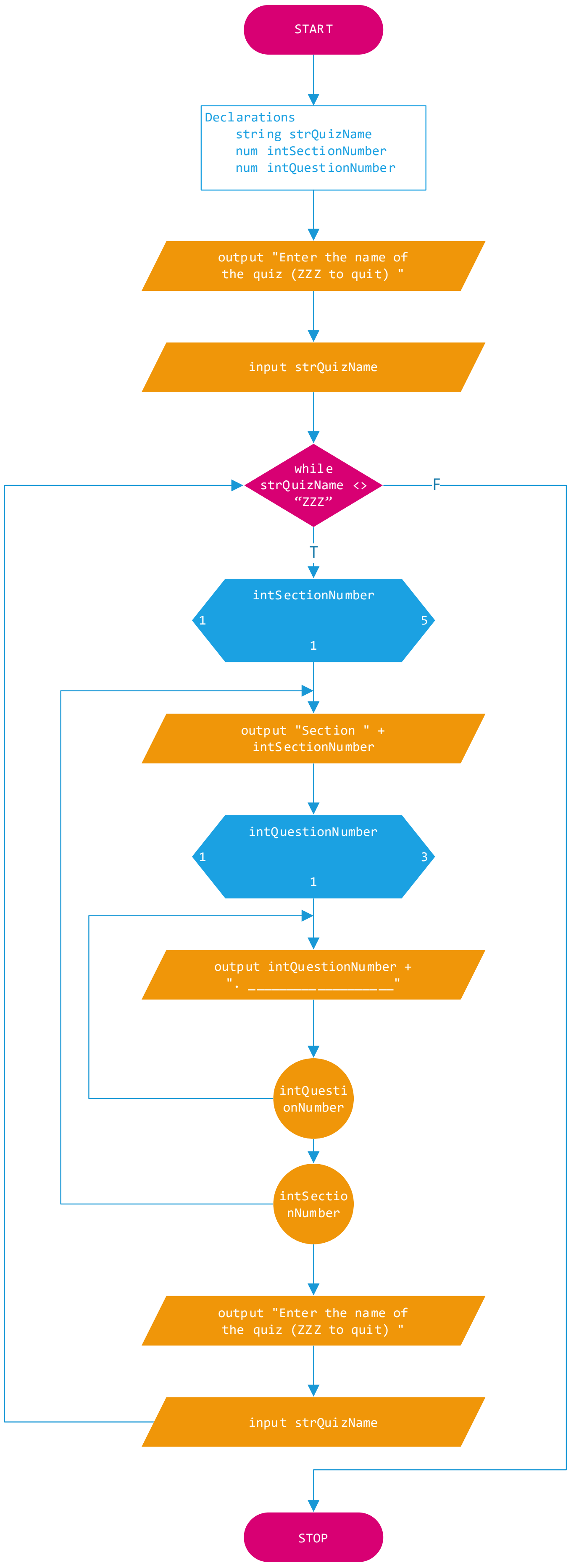
-- We should relook at the code and consider using nested for..loops for the printing of the answer sheets. The reason for the for..loops for controlling the section and question numbers is because we know the number of times these loops will be executed (given date in the storyline shard). We need to realize that we will still be using a while..loop for the quiz names since the user will enter the name of the quiz and choose to end the program with a sentinel value.

-- The following code is using the nested for..loops but will also leave the modules out and code the entire program in the mainline logic (main program).

**pseudocode

```
0. Start
1. Declarations
        string strQuizName
        num intSectionNumber
        num intQuestionNumber

2.      output "Enter the name of the quiz (ZZZ to quit) "
3.      input strQuizName

4.      while strQuizName <> "ZZZ"
                for intSectionNumber = 1 to 5 Step 1
                        output "Section " + intSectionNumber
                        for intQuestionNumber = 1 to 3 Step 1
                                output intQuestionNumber + ". ___________________"
                        endfor
                endfor
                output "Enter the name of the quiz (ZZZ to quit) "
                input strQuizName
        endwhile
5. Stop
```

**flowchart


START
Declarations
string strQuizName
num intSectionNumber
num intQuestionNumber
output "Enter the name of the quiz (ZZZ to quit) "
input strQuizName
while strQuizName <> “ZZZ”
F
T
intSectionNumber
1
5
1
output "Section " + intSectionNumber
intQuestionNumber
1
3
1
output intQuestionNumber + ". ___________________"
intQuestionNumber
intSectionNumber
output "Enter the name of the quiz (ZZZ to quit) "
input strQuizName
STOP

-- Nested loops never overlap.
-- The inner loop is always contained inside the outer loop.
-- The inner loop must always complete all iterations each time it is dealt with, and the outer loop iterates just once.

==
The advantages of looping
==
-- We can code one set of instructions and then repeat these instructions

- This allows the programming to be more efficient
- Less time is required to design and code a complete solution
- This can lead to fewer errors because we are not coding the same instructions multiple times
- The time it takes to compile the program (using the compiler) would be shorter

-- Consider the following: **Imagine being provided with a list of transactions that we are required to go through and perform the same tasks over and over again for each transaction.**

The following example does not show a complete solution. It is an opportunity to understand the need for a loop and relates to the story shared above.

** pseudocode

```
0. Start
1. Declarations
...

2.      houseKeeping()

3.      while not eof
                detailLoop()
        endwhile

4.      endOfJob()
5. Stop
```

- **houseKeeping()** - would initiate access to this file of transactions and set any variables with values that we are required to begin with as we proceed to read through the file.

- the condition for the while..loop is relying on the fact that there are still transactions to be read & when we do reach the end and there are no more transactions, then the loop will stop executing the code within it.

- **detailLoop()** - the code that requires the transaction data and then performs a list of instructions based on the requirements of the story - this code will be repeated for each of the transactions in the file.

- **endOfJob()** - this module would close access to the file and execute code that will end the processing of the story, likely to also be used to output summary information that was gathered while processing each transaction.

==
The logic of a loop
==
-- Provide a starting value for the variable that controls the loop
-- Test the loop control variable to determine whether the instructions within the loop is executed
-- Alter the loop control variable

-- Please remember that the 1st point is done first before entering the loop. This means that it is only executed once at the beginning of the program.

-- The second 2 points is the loop and the code within and it is important to remember that when the program is inside the loop - it will remain inside until the controlling variable contains a value that alters the condition to allow one to leave and proceed to execute the remaining instructions after the loop.

==
Avoiding common loop mistakes
==

-- *Neglecting to initialize the loop control variable*

- If we forget to initialize the variable that controls access to the loop, we may never get the chance to go into the loop, to execute the instructions within the loop.
- It is also possible that because we may forget to initialize, we might get into the loop, but then the instructions that follow might produce a logical error, because of one variable not beginning with the correct value

-- *Neglecting to alter the loop control variable*

- We must always remember that when we are inside a looping structure, we may not leave until the variable that is evaluated in the condition has the value that allows us to exit.
- This would create an infinite loop (one that cannot terminate - we would have to crash the process in the Task Manager or from within the software environment to kill the program, since there is no way out)

-- *Using the wrong comparison with the condition of the loop*

- For example, imagine we need the loop to run a total of 5 times, the code/logical expression

```
intCounter = 1
while intCount < 5 is quite different compared to while intCount <= 5

< 5 - would only run 4 times and
<= 5 - would have been the correct version
```

- The development of the logical expression follows the same rules as we had learnt when we discussed decision logic

-- *Including statements inside the loop that belong outside the loop*

- Using the story of calculating the average age of students in a class.
  - The following code:

```
fltAverage = intSum / intCount
output "The average is " + fltAverage
```

  - if the above 2 lines of code is inside the loop, that would mean that I will be printing out the average after accumulating each of the ages and counting each student in the list. This does not make sense, since I have not fully counted everyone in the class or accumulated everyone's ages while we are still in the loop.

- Another example, imagine a different discount percentage needed to be applied to various sales, and we made the mistake of hard coding the discount and not requesting and coding the program so that the right discount percentage was applied, this would mean that for every calculation, the same discount is applied and while the output statements would look correct, upon further investigation, we would notice the logical error.

-- *Writing the instructions in the wrong order inside a loop*

- In the same way we have learnt about the order of instructions when dealing with simple sequential logic programs, it is important to remember those lessons, so that the code that is executed inside the loop is also executed in the correct order for each iteration of the loop.

-- Many of these mistakes can lead to logical errors, sometimes even run-time errors. It is important to assess our logic with varying examples of data so that we are certain that our code will run for every possibility and print all the required answers as expected.

==

Common loop applications

==

-- To accumulate totals *(e.g., sales total)*

-- To count a number of items *(e.g., the number of students in a class)*

-- Print summary reports

- After a loop has executed, the final values can be used and outputted as a form of a summary of the work completed *(e.g., calculate & output the average, output the highest and lowest marks)*

-- Use a loop to validate data

- If there is an error with the initial data entered, continue asking the user for the correct data until they enter the required information
- Can also be used to validate the correct data type *(e.g., if we request a number and the user enters a string value, or if a name and surname is expected, but the string value does not contain a space)*

-- Use a loop to limit the number of times we prompt the user for information

- If the user enters the incorrect password 3 times (for a login), then block the user's account

----

The similarities and differences between selections and loops

----

-- The textbook and the slides wish that you have a clear understanding of the difference between decision logic and loop logic. More specifically the if..then statement compared to the while..loop.

-- Remember that both use a condition (a logical expression) to determine the path of execution.

-- When drawing the flowchart, it is more evident that after the if..then statement is executed that both the true and false flowlines meet together before going to the next available instruction, whereas the while loop has a flowline for true that enters the loop and a flowline that goes back up to the condition for the repetition and a separate false flowline that leaves the loop. The true and false flowlines do not meet together when the next instruction needs to be executed.