

N5



National 5

Software Design & Development



Learning Intentions

The following provides details of skills, knowledge and understanding that can be covered in the course assignment and the exam.

Development methodologies – Page 3

Describe and implement the phases of an iterative development process: analysis, design, implementation, testing, documentation, and evaluation, within general programming problem-solving.

Analysis – Page 4 -6

Identify the purpose and functional requirements of a problem that relates to the design and implementation at this level, in terms of:

- inputs
- processes
- outputs

Design – Page 7 -10

Identify the data types and structures required for a problem that relates to the implementation at this level, as listed below.

Describe, identify, and be able to read and understand:

- structure diagrams
- flowcharts
- pseudocode

Implementation (data types and structures) – Page 11 -14

Describe, exemplify, and implement appropriately the following data types and structures:

- character
- string
- numeric (integer and real)
- Boolean
- 1-D arrays

Implementation (computational constructs) – Page 15 -26

Describe, exemplify, and implement the appropriate constructs in a high-level (textual) language:

- expressions to assign values
- expressions to return values using arithmetic operations (addition, subtraction, multiplication, division, and exponentiation)
- expressions to concatenate strings
- selection constructs using simple conditional statements with $<$, $>$, \leq , \geq , $=$, \neq operators
- selection constructs using complex conditional statements

- logical operators (AND, OR, NOT)
- iteration and repetition using fixed and conditional loops predefined functions (with parameters):
 - random
 - round
 - length

Implementation (algorithm specification) – Page 27 - 34

Read and explain code that makes use of the above constructs.

Describe, exemplify, and implement standard algorithms:

- input validation
- running total within loop
- traversing a 1-D array - [Page 41](#)

Testing – Page 43 - 48

Describe, identify, exemplify, and implement normal, extreme, and exceptional test data for a specific problem, using a test table.

Describe and identify syntax, execution, and logic errors.

Evaluation – Page 48 - 53

Describe, identify, and exemplify the evaluation of a solution in terms of:

- fitness for purpose
- efficient use of coding constructs
- robustness
- readability:
 - internal commentary
 - meaningful identifiers
 - indentation
 - white space

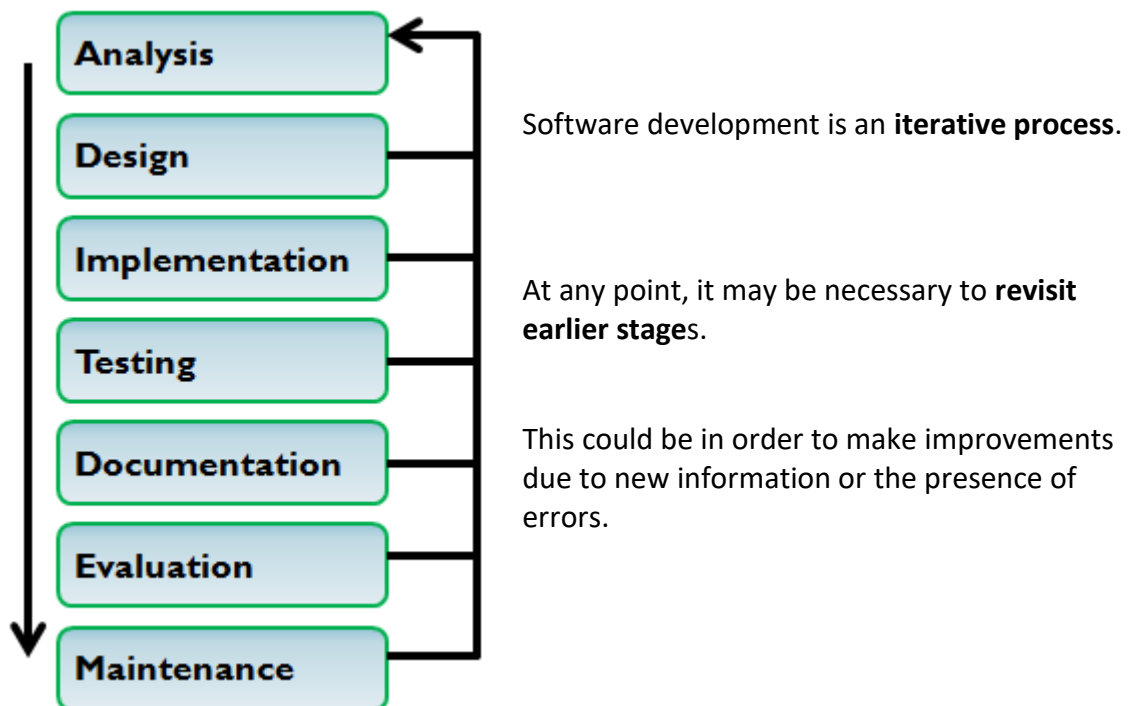
DEVELOPMENT METHODOLOGIES

Waterfall Model

In the Waterfall Model, software is developed in a sequence of stages.

Each stage takes information from the previous stage and provides information to the next.

There are seven stages to this software development process.



ANALYSIS

The purpose of the Analysis stage is to investigate exactly **what the software is supposed to do**.

What does the **client** want the software to do?
Who will use the software?
What are the inputs, processes and outputs?
What hardware will the software uses?

Why? What?
When? Who?
Where?

It is important to read carefully what is required so that you have **clear understanding** of the problem.

The **Systems Analyst** is responsible for gathering information by:

- Visiting and observing client's workplace
- Interviewing client's employees
- Studying documentation
- Issuing questionnaires



The result of the analysis stage is a document called the **Software Specification**.

The Software Specification details the exact requirements of the software to be produced.

The client will read the document and, if agreement is reached, both client and developer will **sign it**.

The Software Specification then becomes a **legally binding document** which can be used to resolve any disputes that may arise between the parties involved.

The analysis stage should also help you to identify the program's **functional requirements**. The functional requirements include identifying the program's:

- **Inputs**
- **Processes**
- **Outputs**

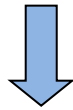
Example

Read the problem below. What are the important pieces of information?

Problem Specification

A program is required that will calculate the total score a student achieved in three of their exams. The students will sit exams in Physics, Computing and Maths.

The total score will be found by adding the scores in each subject. The program should display the overall total score with a suitable message.



Problem Specification

*A program is required that will calculate the **total score** a student achieved in **three** of their **exams**. The students will sit exams in **Physics, Computing and Maths**.*

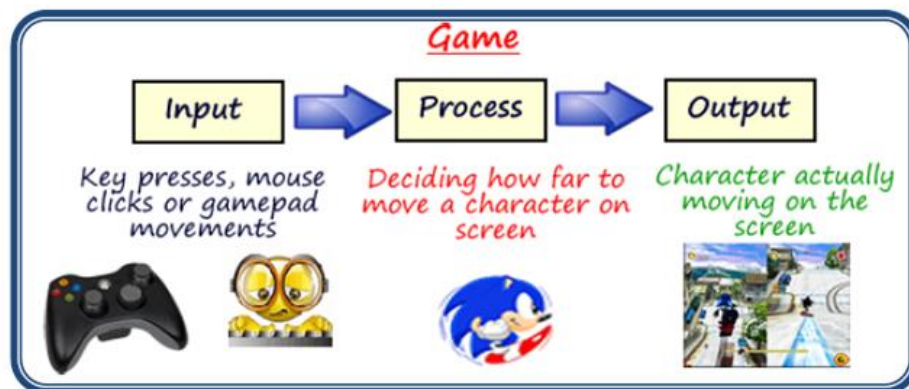
*The total score will be found by **adding the scores** in each subject. The program should **display** the overall **total score** with a **suitable message**.*

Inputs, Processes, Outputs

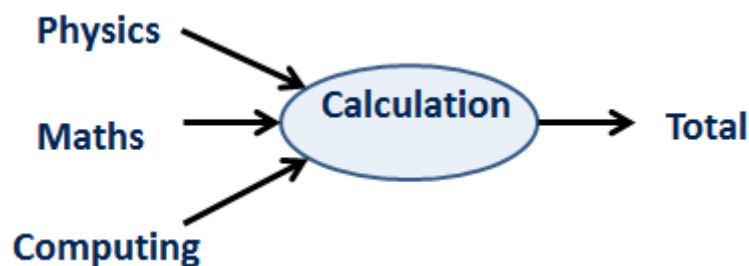
Inputs are data items that must be entered by the user. Information we have to ask the user for. This is the data that the program will take in.

Processes are the things the program will do with the data items. Calculations, formatting etc. are processes.

Outputs are the data items that will be displayed by our program. This will usually be the result of what the program is supposed to do.



Another Example



The **inputs** here are the three exam marks, the **process** is the calculation of the total mark and the **output** is the total mark.

Identify Data Items

A **Data items** is needed for **each** input and output.

You may also need a data item to store each stage of a process (if the process is quite complicated).

It is also important to identify the **type** of data (text, integer, decimal)

Example

Data Item	Data Type
Physics Mark	Integer
Maths Mark	Integer
Computing Mark	Integer
Total Mark	Integer

DESIGN

During the Design stage, a plan for how to solve the problem, as described in the *software specification*, is created. The design stage differs from the analysis stage because analysis is about understanding the problem whereas design is about **How?** you will solve the problem.

Why spend time on design?

The design stage is an extremely important part of software development.

If the program is not designed correctly, then it probably won't do what it is supposed to.

Spending time on a good design makes it easier to write a program.

Decomposition

It can be difficult to know where to start with a large problem that has many tasks to be performed.



Decomposing a problem involves breaking it down into smaller stages that are easier to understand.

It is important to have clear steps for solving a problem that can be easily explained to a computer.

Design Notations

A design notation is a method of listing the steps involved in solving a problem.

Design notations can be graphical:

- **Flow Chart**
- **Structure Diagram**

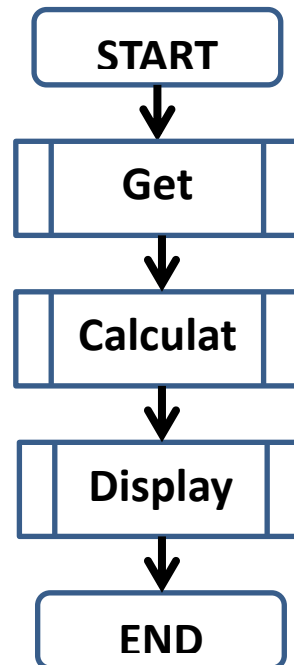
Or text based:

- **Pseudocode**

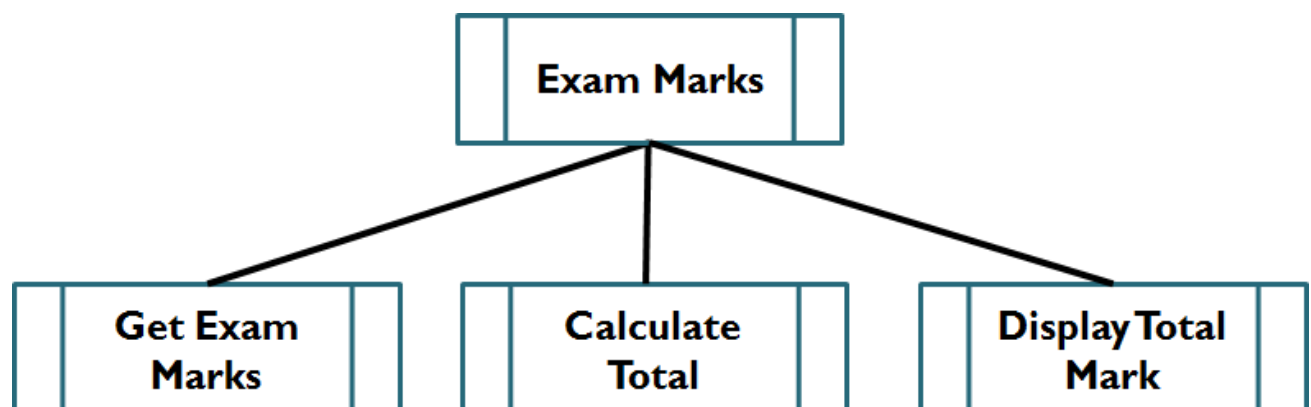
The advantage of graphical design notations over text based design notations is that they give a **visual representation** of the program structure and **order of events**. This gives a **clear overview** of the design and shows the flow of data making it easier to understand.

Flow Chart (graphical)

This type of diagram is read from start to end
Steps are carried in order by following the direction of arrows.



Structure Diagram (graphical)



This type of diagram is read from left to right.
Steps are carried out in order by reading from left to right.

Pseudocode (text based)

Clear English statements are used to describe the steps of the program. Steps are carried out in sequence.

```
RECEIVE physics FROM (Integer) Keyboard  
RECEIVE maths FROM (Integer) Keyboard  
RECEIVE computing FROM (Integer) Keyboard  
  
SET total TO physics + maths + computing  
  
SEND ["The total is ":total] TO DISPLAY
```

Algorithm

Having identified the main steps required for our program, we can say that we have developed an **algorithm**.

An algorithm is a list of step-by-step instructions for solving a problem.

```
RECEIVE physics FROM (integer) Keyboard  
RECEIVE maths FROM (integer) Keyboard  
RECEIVE computing FROM (integer)  
Keyboard  
SET total TO physics + maths + computing
```

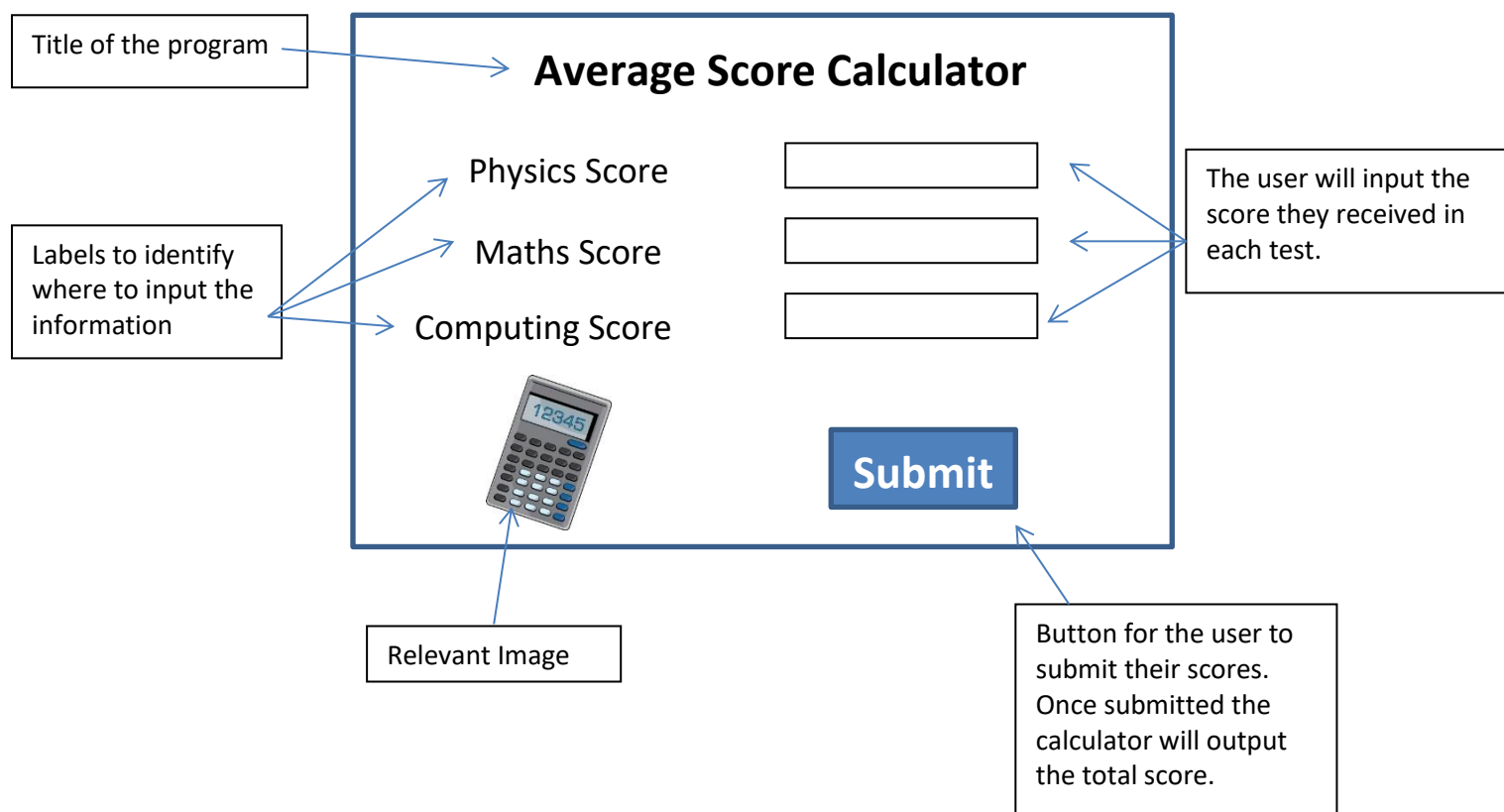
The steps in an algorithm can now be followed and easily converted into lines of programming code (e.g. Visual Basic).

Wireframe

The design of the user interface (the visual layout that allows the user to interact with the programming code) can be represented using a **wireframe** diagram.

A wireframe diagram is a visual representation of how the user interface will look and it will show the position of different elements such as text, graphics, navigation etc. It is also used as a visual representation to demonstrate the input and output of a program.

A wireframe diagram can be a detailed sketch or detailed image as shown below.



Implementation

The implementation stage is where the programming actually takes place.

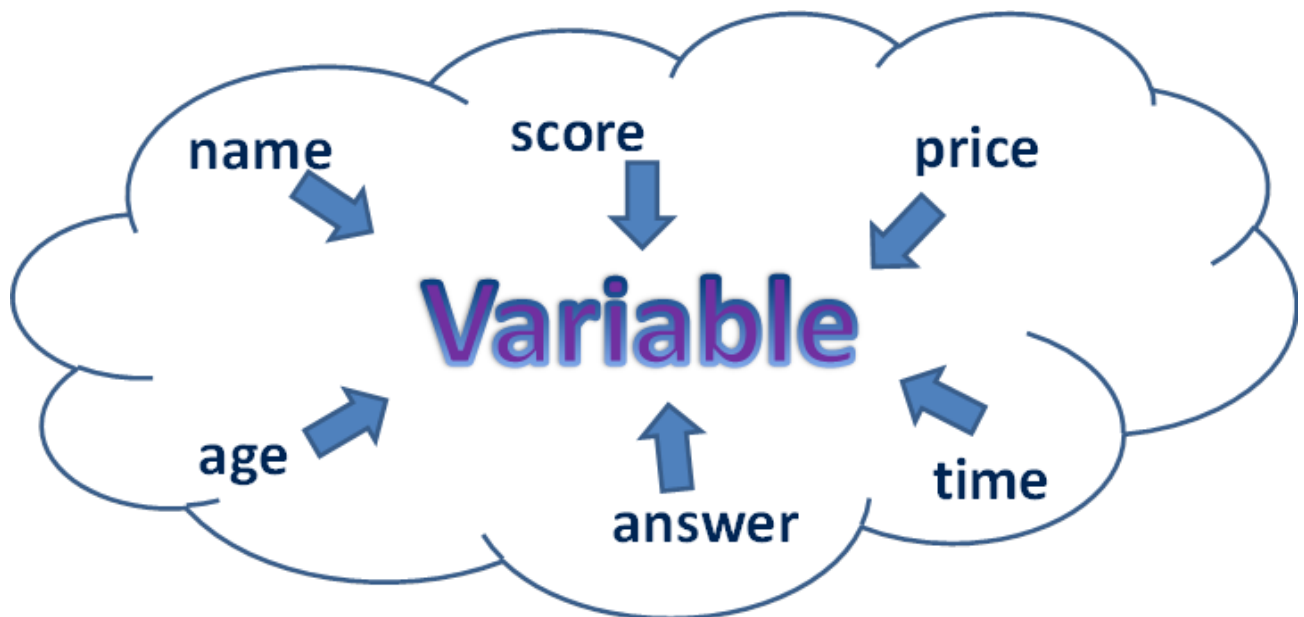
Develop.....

The user interface for the program is created and Design documentation is used and converted into high level language instructions.

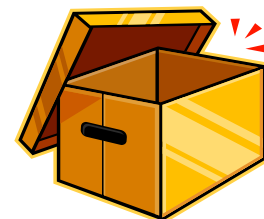
```
public class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024]; string input, stringData;
        TcpClient server;
        try{
            server = new TcpClient(" . . . .", port);
        }catch (SocketException){
            Console.WriteLine("Unable to connect to server");
            return;
        }
        NetworkStream ns = server.GetStream();
        int recv = ns.Read(data, 0, data.Length);
        stringData = Encoding.
        ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
        while(true){
            input = Console.ReadLine();
            if (input == "exit") break;
            newchild.Properties["ou"].Add
            ("Auditing Department");
            newchild.CommitChanges();
            newchild.Close();
        }
    }
}
```

What is a variable?

A variable is used to store a **single item** of **data** in a program.



Imagine a variable as being like a **box** that you can only keep **one** thing in at a time.



Creating a new variable is called **declaring**.

Each variable must be given a **meaningful identifier** (name). Something that tells you what sort of thing it stores.

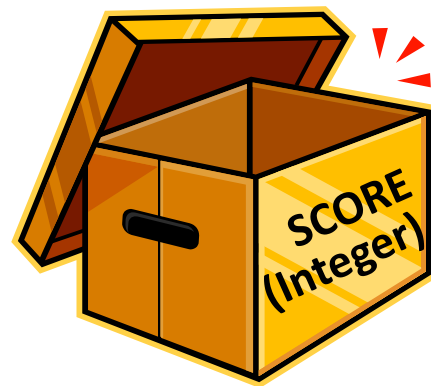
*This variable is used to store a player's score in a game so we'll call it **score**.*



Each variable also has to have a **data type**.

This decides whether the variable can store text or numbers.

*The variable score is being used to hold a whole number so it is declared as an **integer** data type.*



Data Types

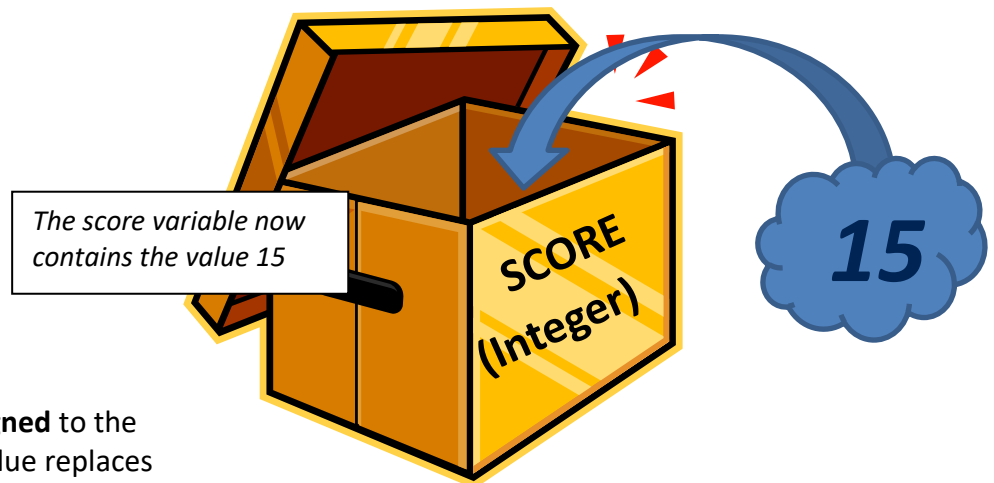
There are five main data types you need to know about:

Data Type	Contents	Example
CHARACTER	Single Letter	"A", "B", "C"
STRING	Several letters	"Fred", "Glasgow"
INTEGER	Whole Number	2, 15, 18, 100
SINGLE (REAL)	Real Number	2.45, 3.9, 12.994
BOOLEAN	True/False	TRUE, FALSE
1-D ARRAYS	List of items	Pupil_name(20)

How do variables work?

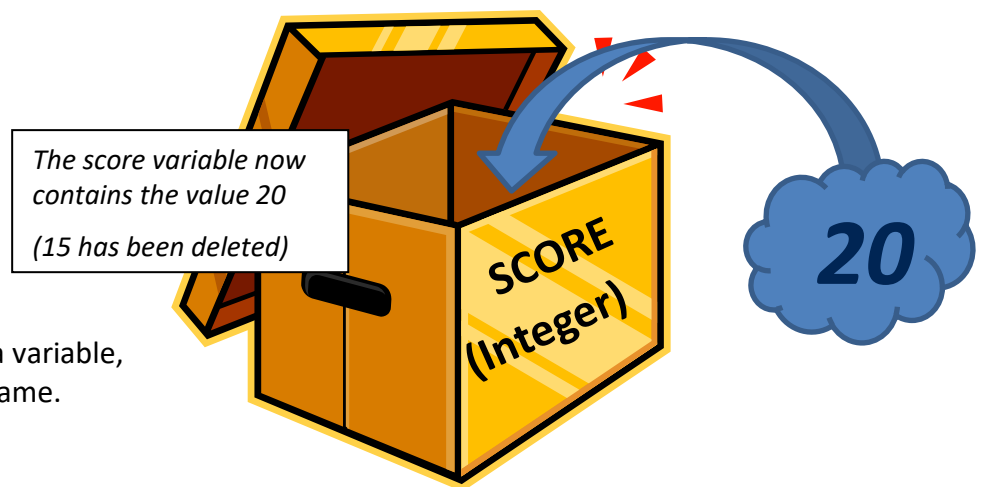
When a value is **assigned** to a variable, this is like putting something into the box.

SET *score* **TO** 15



If a new value is now **assigned** to the same variable, the new value replaces (overwrites) what is already there.

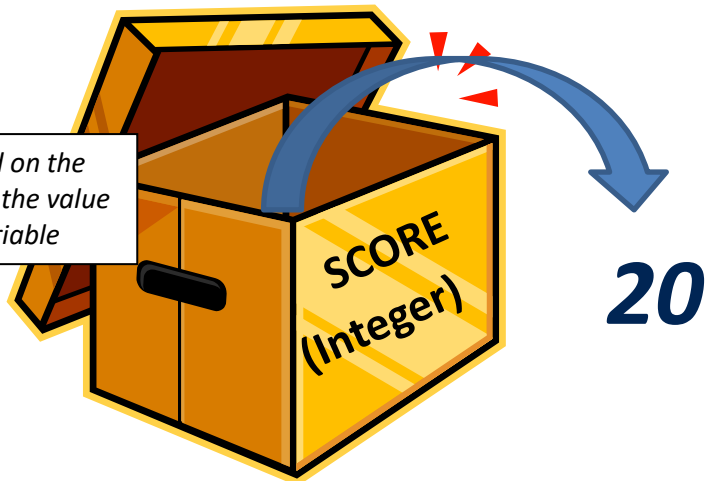
SET *score* **TO** 20



To display the contents of a variable, we can use the variable's name.

SEND *score* TO DISPLAY

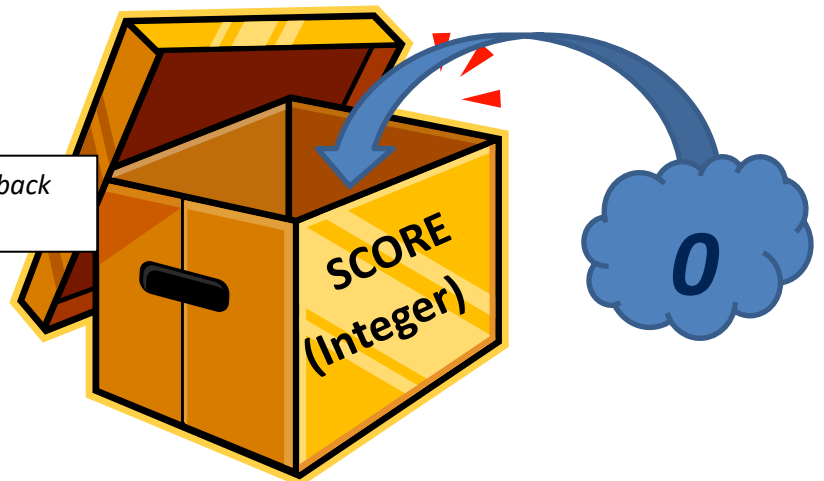
20 would be displayed on the screen because this is the value that is in the score variable



Variables can be reset (cleared) by **initialising** them. It is good practice to do this at the start of a program.

SET *score* TO 0

Score is now set back to 0



A new variable should be **declared** for each piece of information you need to store in your program.

Example

How many variables are required for this program?

A program asks 100m sprinters for their name and times in two heats. It then works out their best time from the heats.

- *Runner_name*
- *Heat1_time*
- *Heat2_time*
- *Best_time*

Variable terms

Declaring: Creating a new variable and setting its name and data type.

Assigning: Placing a value in a variable

Initialising: Resetting or clearing the contents of a variable

Expressions

Expressions in programming are lines of code that carry out a calculation and assign values to variables.

Expressions change the values of variables. You can normally recognise an expression in a programming language because it will a **line of code containing an equals (=) symbol**.

Expressions to Assign Values to Variables

Expressions are used to assign a value to a variable. This could be a newly created variable that is being **initialised**.

SET *total* TO 0

SET *firstname* TO “ ”

SET *price* TO 0.00

SET *found* TO FALSE

Or a variable that already contains a value and is being changed.

SET *total* TO 15

SET *firstname* TO “Fred”

SET *price* **TO** 2.49

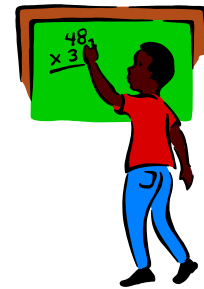
SET *found* **TO** TRUE

Expressions to Return Values using Arithmetic Operations

Arithmetic Operations are simply calculations that are performed within a program.

The simplest example of an arithmetic operation in a program is adding two numbers together.

$$2 + 2 = 4$$



Arithmetic Operations that can be performed in a program are:

- Addition
- Subtraction
- Multiplication
- Division
- Exponent

The result of an arithmetic operation expression is **returned** and usually gets **assigned** to a **variable** so that the variable stores the answer to the calculation.

SET *answer* **TO** $2 + 2$

The answer variable will now store the value 4

Variables can also be used as part of the arithmetic operation.

RECEIVE *num1* **FROM** (*Integer*) **KEYBOARD**
RECEIVE *num2* **FROM** (*Integer*) **KEYBOARD**

The user enters two numbers, stored in variables

The answer variable stores the result of the expression

SET *answer* **TO** $\text{num1} + \text{num2}$

For addition, the + symbol is used as in maths. Subtraction also uses the - symbol from maths. However the other operations use different symbols.

Operation	Symbol	Example
Addition	+	SET sum TO num1 + num2
Subtraction	-	SET difference TO num1 - num2
Multiplication	*	SET product TO num1 * num2
Division	/	SET quotient TO num1 / num2
Exponent	^	SET power TO num1 ^ num2

Examples

SET answer TO num1 + num2

SET answer TO num1 - num2

SET answer TO num1 * num2

SET answer TO num1 / num2

SET answer TO num1 ^ num2

Concatenation

Concatenation is the joining together of two or more variables or the joining together of text and a variable. The ampersand (&) symbol is used to represent concatenation.

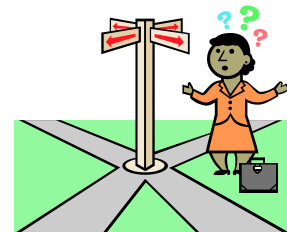
Examples

SEND "Answer is:" & answer TO DISPLAY

SEND "Hello:" & firstName & surName TO DISPLAY

What is Selection?

Selection constructs are used in a program to allow it to ask a question and take a different path depending on the answer.



How do you decide each day whether you have to go to school or not? When you wake up in the morning, the rule you use might be:



IF today is a weekday **THEN**
go to school

The commands we would use in our algorithm are very similar to this rule.

1. **IF** *today* = “weekday” **THEN**

Line 1 is our condition

2. Go to school

Line 2 is **only** carried out if the condition is true

3. **END IF**

4. ...

The commands between **IF** and **END IF** will only be carried out if the condition is true.

In the example above, we only go to school if the condition in line 1 is true. If the condition is false, we do nothing other than move on to line 4.

We could decide that, on a day when we don't go to school, we always go shopping.

Now, our morning rule might be:

IF today is a weekday **THEN**
 go to school
OR ELSE
 go to the shops



We use the **ELSE** command to indicate what should happen when it is not a weekday.

1. **IF** *today* = "weekday" **THEN**

Line 1 is our condition

2. Go to school

Line 2 is **only** carried out if the condition is **true**

3. **ELSE**

4. Go shopping

Line 4 is **only** carried out if the condition is **false**

5. **END IF**

6. ...

The program then moves on to line 6 which is definitely carried out.

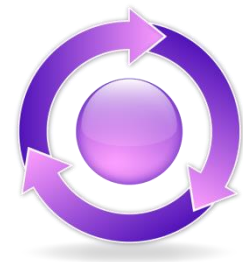
Simple conditions can use the following operations:

Operation	Symbol	Example
Less than	<	num1 < num2
Greater than	>	num1 > num2
Less than or equal to	<=	num1 <= num2
Greater than or equal to	>=	num1 >= num2
Equal	=	num1 = num2
Not Equal to	<>	num1 <> num2

What is Iteration?

Iteration or repetition is the process of **repeating** instructions in a program a desired number of times.

Iteration is achieved in programming using **loops**.



Using loops in a program can drastically reduce the number of lines of code you have to type.

Fixed Loops

Imagine you wanted to tell someone to walk up and down the stairs 5 times – but you can only issue **one** instruction at a time.

1. Walk **UP** stairs
2. Walk **DOWN** stairs
3. Walk **UP** stairs
4. Walk **DOWN** stairs
5. Walk **UP** stairs
6. Walk **DOWN** stairs
7. Walk **UP** stairs
8. Walk **DOWN** stairs
9. Walk **UP** stairs
10. Walk **DOWN** stairs



There are actually only **two** commands here that are repeated over and over. What are they?

A better method for this type of scenario is to use a loop and place the repeat instructions inside it.

1. **REPEAT 5 TIMES**
2. Walk **UP** stairs
3. Walk **DOWN** stairs
4. **END REPEAT**

These instructions
will repeat exactly 5
times

Conditional Loops

What if we wanted to ask someone to walk up and down the stairs **until lunch time**? How many times will they do it? Do we know?

For this scenario, we don't know exactly how many times our friend will be able to walk up and down the stairs.



It could be 5 times, 50 times, 200 times or more.

A **Conditional Loop** allows us to repeat instructions **until** a particular event (condition) occurs in our program.

1. **REPEAT**
2. Walk **UP** stairs
3. Walk **DOWN** stairs
4. **UNTIL** **time** = 12:30

These instructions
will be repeated

DO WHILE Conditional Loop

This loop starts repeating instructions. It checks the condition at the **end** of the loop. This means that the repeated instructions will always be carried out **at least once**.

1. **REPEAT**

- 2. Walk **UP** stairs
- 3. Walk **DOWN** stairs

These instructions
will be repeated

4. **UNTIL** *time* = 12:30

This means that the repeated instructions will always be carried out **at least once**.

WHILE Conditional Loop

This loop checks the condition at the start of the loop. It then starts repeating instructions.

1. **WHILE** *time* <> 12:30 **DO**

- 2. Walk **UP** stairs
- 3. Walk **DOWN** stairs

These instructions
will be repeated

4. **END WHILE**

This means that the repeated instructions may never run at all (if it is already 12:30 at the start of the loop).

What are Logical Operations?

Logical Operations are used to create **complex conditions**.

Complex conditions are conditions that have **two or more** parts to them.



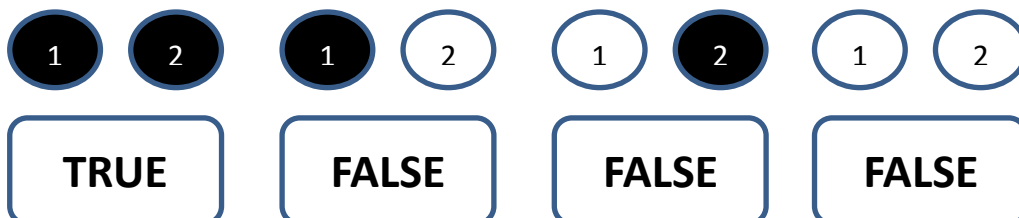
The main logical operators are:

- **AND**
- **OR**
- **NOT**

How do complex conditions work?

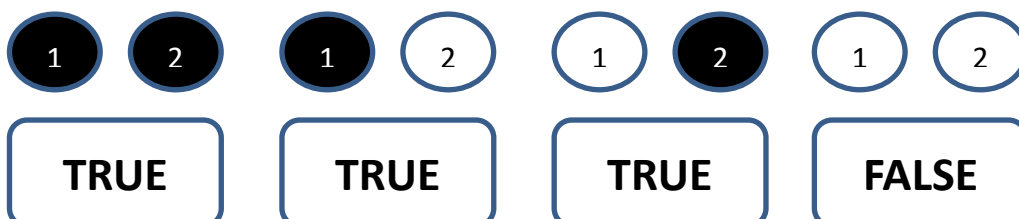
AND checks that both parts of a condition is true

circle1 = black **AND** circle2 = black



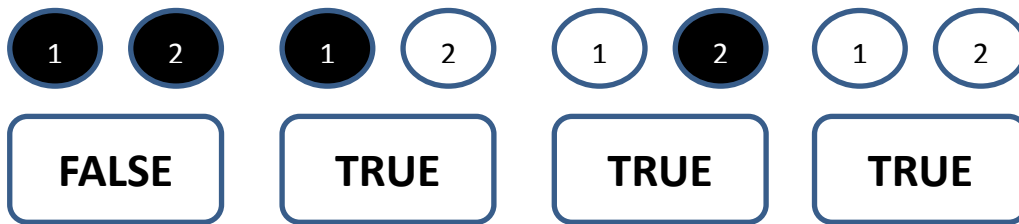
OR checks that either part of a condition is true

circle1 = black **OR** circle2 = black

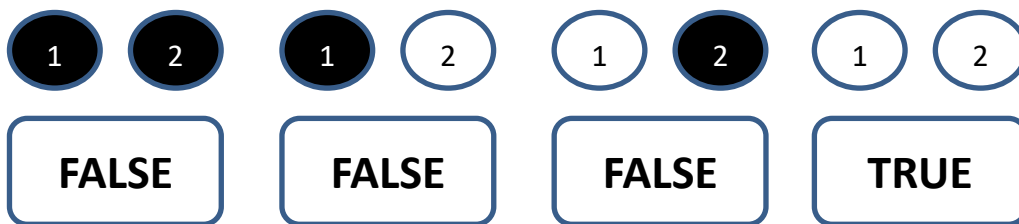


NOT gives the opposite answer

NOT (circle1 = black **AND** circle2 = black)



NOT (circle1 = black **OR** circle2 = black)



Logical Operations are normally used in **IF** statements or **Conditional Loops**.

IF *condition 1* **AND** *condition 2*

IF *condition 1* **OR** *condition 2*

IF NOT (*condition 1* **OR** *condition 2*)

UNTIL *condition 1* **AND** *condition 2*

UNTIL *condition 1* **OR** *condition 2*

UNTIL NOT (*condition 1* **OR** *condition 2*)

Examples:

IF *num1* > 3 OR *num2* > 13 THEN

IF *num1* > 2 AND *num2* < 13 THEN

IF *answer* = "Yes" OR *answer* = "No" THEN

IF NOT (*num1* = 5 AND *num2* < 10) THEN

UNTIL *first* = "Fred" AND *second* = "Jones"

UNTIL *answer* = "Yes" OR *answer* = "No"

Standard Algorithms

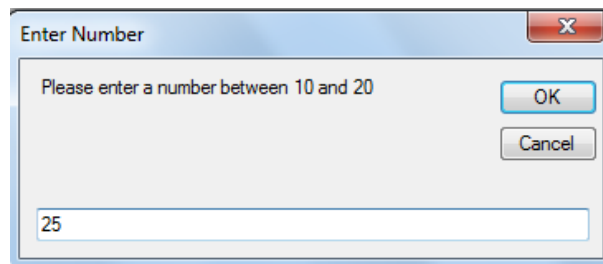
Input Validation

Input Validation is a **Standard Algorithm** that can be used in **any** program.

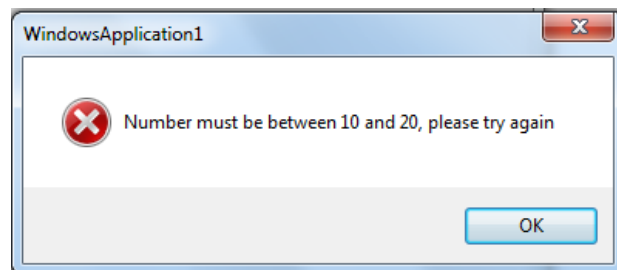
The purpose of Input Validation is to check that a user has entered data that is in the format that was expected.



Imagine your program asks the user to enter a number between 10 and 20.



If the user enters a value out with this range, the Input Validation algorithm will keep asking them to re-enter until they enter a valid number.



How does Input Validation work?

Let's look at the 2 ways this can be implemented in Visual Basic

Do ... Loop until	
Do	
Userinput = InputBox("Please enter data")	
IF Userinput < 10 OR Userinput > 20 then	
MsgBox ("Invalid input")	
End if	Optional
Loop Until UserInput >= 10 AND UserInput <= 20	
Do ... While Loop * Most Efficient way *	
Userinput = InputBox("Please enter data")	
Do While UserInput < 10 OR UserInput > 20	
Userinput = InputBox("Please re-enter valid data")	
Loop	

Input Validation using a WHILE Loop (Algorithm)

RECEIVE number FROM KEYBOARD

WHILE number is out of range

SEND error message TO DISPLAY

RECEIVE number FROM KEYBOARD (Re-enter)

END LOOP

Explaining Input Validation Code – example

User is asked to enter a number between 1 and 10 inclusive.

Line 1	RECEIVE number FROM KEYBOARD
Line 2	WHILE number < 1 OR number > 10
Line 3	<i>SEND error message TO DISPLAY</i>
Line 4	RECEIVE number FROM KEYBOARD (Re-enter)
Line 5	END LOOP

- User is asked to enter a number between 1 and 10
- The conditional loop will execute if the user inputs a number less than 10 or greater than 20. (Valid range = 10 to 20 inclusive)
- An error message will appear on the screen
- The user will be asked to re-enter a valid number. This will continue to loop until they have entered between 10 and 20
- The loop will end once valid data has been entered.

VB6 Code

```
num = InputBox ("Enter number between 1 and 10")  
Do While num < 1 Or num > 10  
    MsgBox "Must be num between 1 and 10"  
    Num = InputBox ("Enter number between 1 and 10")  
Loop
```

Input Validation using a DO Loop Until (Algorithm)

Example VB6 Code

Do

```
Userinput = InputBox("Please enter data")
```

```
IF Userinput < 10 OR Userinput > 20 then  
    MsgBox ("Invalid input")  
End if
```

Optional

```
Loop Until UserInput >= 10 AND UserInput <= 20
```

Example: Collecting month number from user

REPEAT

RECEIVE **month** FROM KEYBOARD

IF **month** <1 OR **month** >12 **THEN**
 SEND error message **TO DISPLAY**
END IF

UNTIL **month** >= 1 AND **month** <=12

A **conditional loop** is used in this algorithm.

It uses the **complex condition** **month** <1 OR **month** > 12 to **validate** the month entered.

This makes sure that the user has entered a value between 1 and 12.

The Input Validation algorithm can be included in any program simply by changing the **conditions** for the inputted data.

Example: Collecting age of secondary school pupils

REPEAT

RECEIVE *age* FROM KEYBOARD

IF *age* <11 OR *age* >18 THEN
 SEND error message TO DISPLAY
END IF

UNTIL *age* >= 11 AND *age* <= 18

A **conditional loop** is used in this algorithm.

It uses the **complex condition** *age* <11 OR *age* > 18 to **validate** the user's *age* entered.

This makes sure that the user has entered a value between 11 and 18.

Explaining Input Validation Code

Example 1: Explain what happens if the user enters the value 10

Line 1	REPEAT
Line 2	RECEIVE <i>age</i> FROM KEYBOARD
Line 3	IF <i>age</i> <11 OR <i>age</i> >18 THEN
Line 4	SEND error message TO DISPLAY
Line 5	END IF
Line 6	UNTIL <i>age</i> >= 11 AND <i>age</i> <= 18

- The **condition** in **line 3** would be evaluated as **true** because 10 is less than 11.
- Therefore **line 4** would be **executed**, displaying an **error message** on screen
- The **condition** in **line 5** would be evaluated as **false** because 10 is not between 11 and 18 so the conditional loop will return to line 1.
- In **line 2** the user would be asked to **re-enter** their age.

Example 2: Explain what happens if the user enters the value 15

```
Line 1    REPEAT
Line 2        RECEIVE age FROM KEYBOARD
Line 3        IF age <11 OR age >18 THEN
Line 4            SEND error message TO DISPLAY
Line 5        END IF
Line 6    UNTIL age >= 11 AND age <= 18
```

- The **condition** in line 3 would be evaluated as **false** because 15 is not less than 11 or greater than 18.
- Therefore **line 4** would not be **executed**

The **condition** in line 5 would be evaluated as **true** because 15 is greater than 11 so the conditional loop would **terminate**

The Input Validation algorithm can also be used for text entry where specific words or characters are expected.

Example: Asking the user to enter Yes or No

```
REPEAT

    RECEIVE response FROM KEYBOARD

    IF response <> "Yes" AND response <> "No" THEN
        SEND error message TO DISPLAY
    END IF

UNTIL response = "Yes" OR response = "No"
```

Logical Operations

The use of logical operators is extremely important in input validation.

When validating a range of numbers

```
IF month < 1 OR month > 12 THEN  
    SEND error message TO DISPLAY  
END IF  
UNTIL month >= 1 AND month <= 12
```

OR here

AND here

When validating for particular text

```
IF response <> "Yes" AND response <> "No" THEN  
    SEND error message TO DISPLAY  
END IF  
UNTIL response = "Yes" OR response = "No"
```

AND here

OR here

Standard Algorithms

The process of keeping a running total is used within a loop structure to keep an ongoing calculation or tally of values.

Running Total using a fixed loop

```
DECLARE total INITIALLY 0
FOR loop FROM 1 TO 10 DO
    RECEIVE number FROM KEYBOARD
    SET total TO total + number
END FOR
```

Example VB6 Code

```
total = 0

For counter = 1 To 5
    points = InputBox("How many points were received?")
    total = total + points
    lstScore.AddItem "Round " & counter & " score = " & points
Next
```

In this example, the user will be asked to enter how many points they received over 5 events. The points will be added to an overall total and then displayed to the screen.

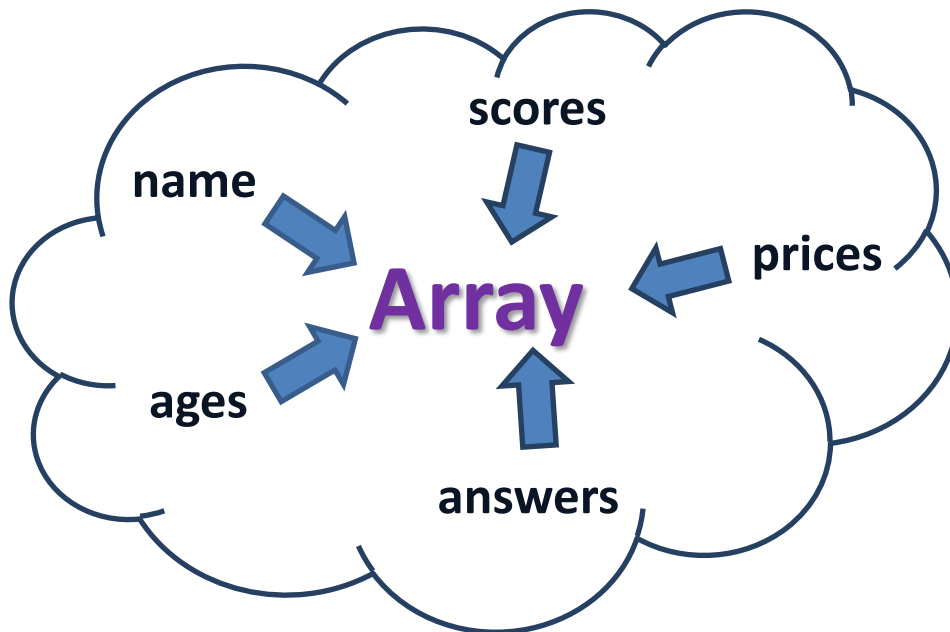
Running total within a loop: example 2 (conditional loop)

This program is used to calculate the sum of an unknown number of values entered by the user one at a time.

```
DECLARE total INITIALLY 0
REPEAT
    RECEIVE number FROM KEYBOARD
    SET total TO total + number
    SEND "Do you wish to enter another value" TO DISPLAY
    RECEIVE choice FROM KEYBOARD
LOOP UNTIL choice = "no"
```


What is a 1D array?

A 1D array is a **data structure**. It is similar to a variable but it can store **several items of data** as long as they are of the **same type**.



Imagine a variable as being like a **list** of items that must all be on the same subject.

Creating a new array is called **declaring**

Like variables, each array must be given a **meaningful identifier** (name), something that tells you what sort of thing it stores.

*This array is used to store pupil ages so we'll call it **ages**.*

Ages
15
16
13
14
12

Each array also has to have a **data type**.

All of the items in an array must have the **same** data type.

*An age is a whole number so we will set this array's data type to **Integer***

(Integer)

Ages

15

16

13

14

12

Each array also has to have its maximum **size** specified.

Each position in the array has an index number to identify it.

*This array can store five items so its size is **5***

(Integer)

Ages

(0)

15

(1)

16

(2)

13

(3)

14

(4)

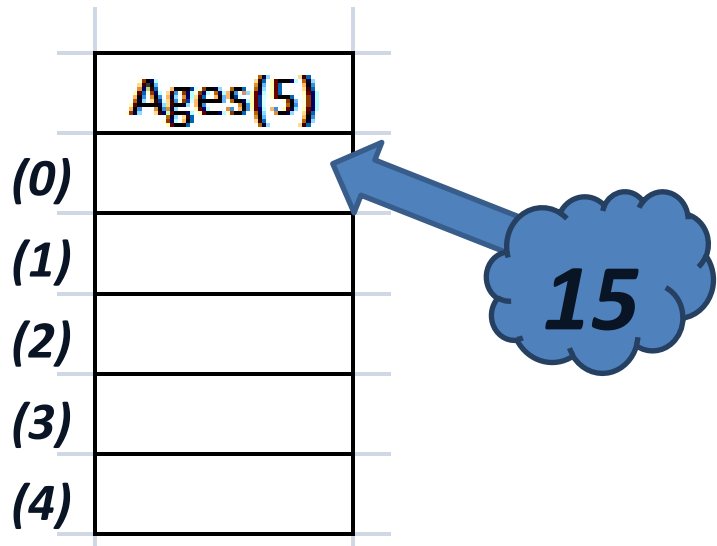
12

How do arrays work?

When a value is **assigned** to an array, the position you want to put it into must also be given.

SET **Ages**(0) **TO** 15

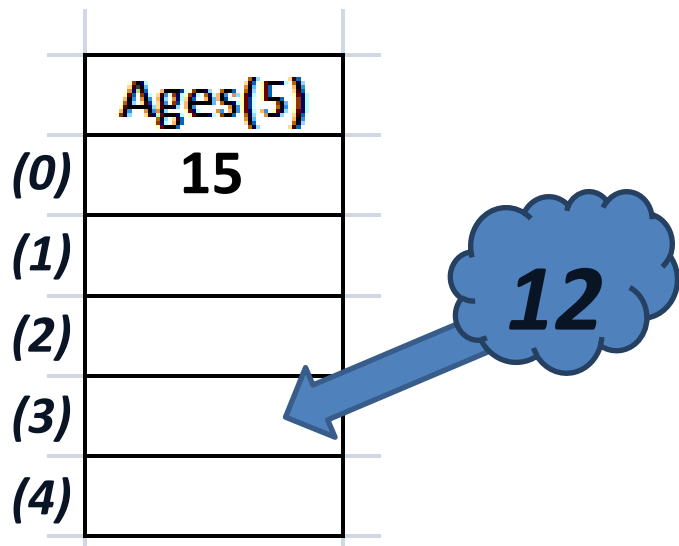
Position 0 of the age array now contains the value 15.



Using a different index number will assign a value to a different position in the array.

SET **Ages**(3) **TO** 12

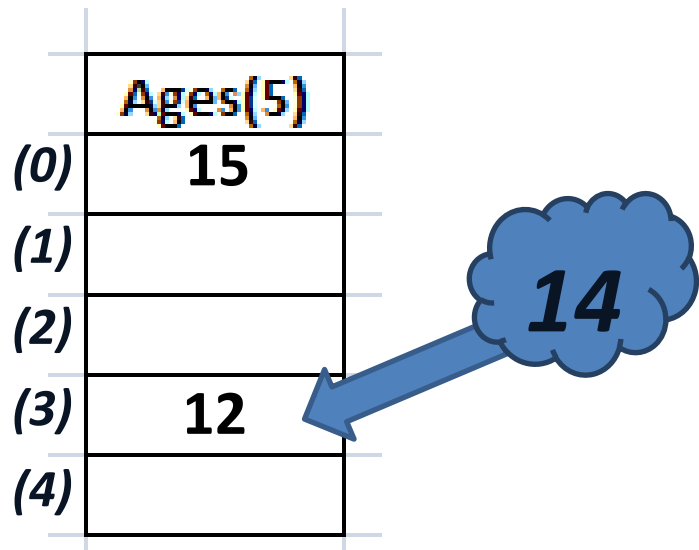
Position 3 of the age array now contains the value 12



If a new value is assigned to a position that already contains a value, that value is overwritten.

SET Ages(3) TO 14

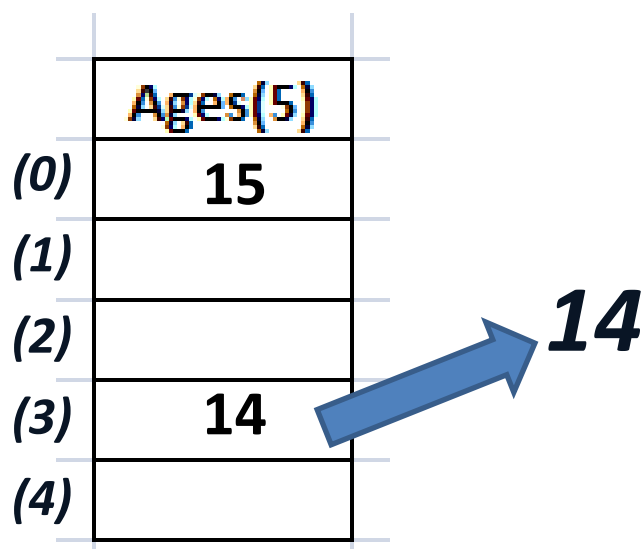
Position 3 of the age array now contains the value 14



To display the contents of an array position, we can use the array's name and the index position.

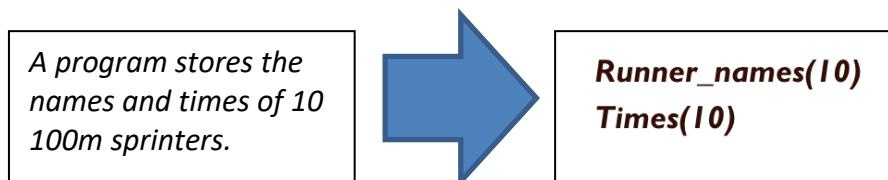
SEND ages(3) TO DISPLAY

14 would be displayed on the screen because that is the value that is currently in position 3 of the ages array



When do you need arrays?

A new array should be **declared** when there are a number of similar items of the same type to be stored.



Working with Arrays

In a large array it may not be practical to enter information into each position individually.

SET *ages*(1) **TO** 12

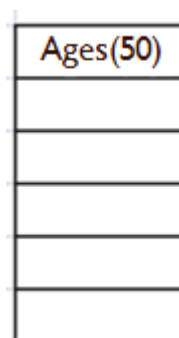
SET *ages*(2) **TO** 14

SET *ages*(3) **TO** 11

...

SET *ages*(50) **TO** 13

of



This array would require 50 lines of code just to fill it.

Using a loop makes life much easier.

FOR EACH *age* **FROM** *ages*

RECEIVE *age* **FROM** (*Integer*) **KEYBOARD**

END FOREACH

For Each starts at position 1 in the array and repeats until it reaches the end of the array.

Loops can also be used when performing operations on each value in an array. The pseudocode below would add up **all** of the ages in the array.

FOR EACH *age* **FROM** *ages*

total = **total** + *age*

END FOREACH

Arrays can be used to allow the user to select an item from the list to be used. The pseudocode below will double the age of the user selected position from the ages array.

RECEIVE *user_value* **FROM** (*Integer*) **KEYBOARD**

SET *doubled* **TO** $2 * \text{ages}(\text{user_value})$ **KEYBOARD**

The user enters an array position which is stored in **user_value**

The **contents** of the **ages** array **position** selected by the user will be used in the calculation.

The value entered by the user must be within the range of the array size (1 to 50 in this case).

Array terms

Declaring: Creating a new array and setting its name and data type.

Assigning: Placing a value in a array position

Initialising: Resetting or clearing the contents of a array

Standard Algorithms

Traversing a 1D array: example 1 (fixed loop)

This program is using a loop to access each element of an array, for the purposes of processing the data in the array.

```
DECLARE allScores INITIALLY [ 12,34,23,54,32,67,26,23 ]
FOR counter FROM 0 TO 7 DO
    IF allScores[counter] >= 50 THEN
        SEND "Great Score" & allScores[counter] TO
        DISPLAY
    END IF
END FOR
```

Traversing a 1D array: example 2 (fixed 'for each' loop with running total included)

This program is using a loop to access each element of an array, for the purposes of processing the data in the array.

```
DECLARE allScores INITIALLY [ 12,34,23,54,32,67,26,23 ]
DECLARE total INITIALLY 0
DECLARE counter INITIALLY 0

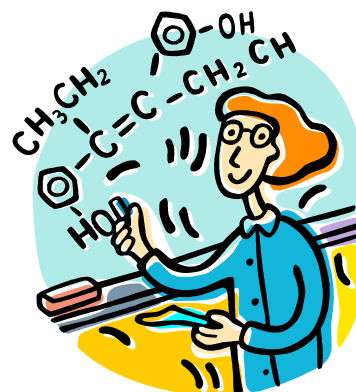
FOR EACH FROM allScores DO
    SET total TO total + allScores[counter]
    SET counter TO counter + 1
END FOR
```

What are Predefined Functions?

Predefined functions are commands that can be used in any program to carry out a **calculation** or **format text and numbers** in a particular way.

They are like **shortcuts** as they save you having to write your own lines of code to carry out the function's task.

There are many predefined functions (too many to list them all) and they vary slightly between different programming languages.



Typical Predefined Functions

Function	Purpose
Math.Round	Round a number to a specified number of decimal places.
Rnd	Generate a random number
Len	Find the length of a string (how many characters it contains)

There are many more functions available.

If there is a particular operation you want to perform in your program, it is always a good idea to look up a good website that will give you a list of available functions for the language you are using.

<http://howtostartprogramming.com/vb-net/>

<http://www.tutorialspoint.com/vb.net>

<http://www.functionx.com/vbnet/>

Testing

Why Test?

Testing is an important stage of the Software Development Process.

It allows the finished program to be verified to ensure that it does what it is supposed to do, doesn't crash and produces correct and reliable results.

Testing helps to find errors in the code before it is released.

It is important that a carefully considered test plan is created. It is **vital** that a test plan is produced **before** the solution is implemented to ensure the software is tested **systematically**.



Test Data

The test plan includes:

- Details of what is to be tested
- Test data values
- Expected outputs
- Type of testing

To ensure that a program is tested comprehensively and thoroughly, three types of test data should be used.

- **Normal**
- **Extreme**
- **Exceptional**



Normal Test Data

Normal test data checks that the program will accept the inputs from the user that it is supposed to.

A program is designed to ask the user to enter a number
between **10** and **20**

Normal data in this case would be:
11,12,13,14,15,16,17,18 and **19**

A program is designed to ask the user to enter a number
between **1** and **100**

Normal data in this case would be:
all numbers from 2 to 99

A program is designed to ask the user to enter either "**Yes**"
or "**No**"

Normal data in this case would be:
Yes, No

Extreme Test Data

Extreme test data checks that the program will accept the inputs from the user that are on the boundaries of what it is acceptable.

A program is designed to ask the user to enter a number
between **10** and **20**

Extreme data in this case would be:
10, 20

A program is designed to ask the user to enter a number
between **1** and **100**

Extreme data in this case would be:
1, 100

A program is designed to ask the user to enter a number
between **0** and **50**

Normal data in this case would be:
0, 50

Exceptional

Exceptional test data checks that the program uses **input validation** so that it will **not accept** inputs from the user that it is not supposed to.

A program is designed to ask the user to enter a number between **10** and **20**

Extceptional data in this case would be:
...7,8,9 and 21,22,23...

A program is designed to ask the user to enter a number between **1** and **100**

Exceptional data in this case would be:
...-2,-1, 0 and 101, 102, 103...

A program is designed to ask the user to enter either **“Yes”** or **“No”**

Exceptional data in this case would be:
“Maybe”, “Perhaps”, 10

Test Plan



A test plan is required to ensure that a program is tested **systematically**.

Systematic testing using a test plan means that you have thought carefully about how you will test different parts of the program.

You will also have a reason for every test you carry out, testing should **not** be random.

Example

This program should accept three test scores between 0 and 50 and calculate the total and average.

Test No.	Reason	Test Data	Expected Result	Actual Result	Test Outcome
1	Normal Test	Score1: 34 Score2: 45 Score3: 29	Total: 108 Average: 36	Total: 108 Average: 36	PASS
2	Normal Test	Score1: 12 Score2: 19 Score3: 2	Total: 33 Average: 11	Total: 33 Average: 11	PASS
3	Extreme Test	Score1: 50 Score2: 50 Score3: 50	Total: 150 Average: 50	Total: 150 Average: 50	PASS
4	Extreme Test	Score1: 0 Score2: 0 Score3: 0	Total: 0 Average: 0	Total: 150 Average: 50	PASS
5	Exceptional Test	Score1: 65 Score2: 52 Score3: 90	Not Accepted	Total: 207 Average: 69	FAIL
6	Exceptional Test	Score1: -1 Score2: -40 Score3: -100	Not Accepted	Not Accepted	PASS

Error Types

Error Type	Description	Example
Syntax Error	The rules of the programming language have been broken. The program will not start.	FOR index <u>IS</u> 1 to 10 • is should be = IF age = 5 <u>THEEN</u> • Theen should be then
Execution Error	Program is asked to do something impossible or illegal. The program will run but will crash.	answer = total / 0 Set age TO "Fred"
Logic Error	Program will run and will not crash. Does not produce the expected results.	SET answer TO 6 * 4 SEND "6 + 4 =" & answer TO DISPLAY • Expected result is 10 but error in the first line means 24 is displayed.

Evaluation

During the Evaluation Stage, the overall success of the entire project is considered.



An evaluation report would discuss:

- **Usability** of the software (easy to use?)
- **Accessibility** of the software
- **Efficiency** of the code in terms of hardware use
- Does the solution meet the Software Specification **requirements**?
- What **improvements** could be made?

Fitness for purpose

Once a program has been analysed, designed, implemented and tested it is important to evaluate the program to ensure it is **fit for purpose**.

Evaluating allows us to ensure the program does the job it was designed to do and think about any improvements that could be made.

When evaluating a program we should ensure that the program:

- Is easily understood
- Is a completed solution (fully solves the problem)
- Is efficient
- Meets the design/specification given

Efficient use of coding constructs

Ensuring our program is efficient is important. An efficient program fully meets the specification while making best use of coding constructs.

Removing unnecessary code, arriving at the final output in the quickest way and use of the correct variables, data types and programming concepts allows for an efficient program.

Robustness

A program is **robust** if it has the ability to cope with errors or incorrect input for the user without the program crashing.

For example, a program has been developed to ask the user to input their age as an integer. However, a user accidentally types their age in as a string. If the program was to crash then it would not be a robust program.

What is readability?

It is important for programs to be written in a readable fashion so that they can be easily understood.

Imagine if you (or someone else) wanted to make changes to a program that **you** wrote a year ago or more.

If you didn't take care to make sure that it could be easily understood then it will probably take a long time to make the changes.

Some methods of making a program readable are:

- Use meaningful identifiers - variable names
- Use internal commentary
- Use structured listings (indentation, capitalisation)
- Use plenty of white space

Meaningful Identifiers - Variable Names

If we were to write a program using variable names such as **X**, **Y** and **Z** then it is difficult to know what values they are intended to store.

Dim x as
Single
Dim y as

Dim total as Single
Dim PupilCost as
Single

It is much better to use sensible variables names like **total**, **PupilCost** and **pupils**.

Internal Commentary

Internal commentary is used to give **descriptions** of what lines or sections of code do.

Internal Commentary is written between the lines of actual code but is **ignored** by the computer when executing the program.

'School Trip Costs

'By J Bloggs

'01/04/2013

'declare variables

Dim total as Single

Dim PupilCost as Single

Dim pupils as Integer

Example

Which of these programs is more readable?

```
For i = 1 To 5
Do
x(i) = InputBox("Please enter test score " & i)
if x(i) < 0 Or x(i) > 100 Then
MsgBox "Score must be between 0 and 100"
End If
Loop Until x(i) >= 0 And x(i) <= 100
Next i
average = (x(1) + x(2) + x(3) + x(4) + x(5)) / 5
```

```
FOR scores = 1 To 5    'loop for each test score
    DO
        tests(scores) = InputBox("Please enter test score " & scores)
        IF tests(scores) < 0 Or tests(scores) > 100 THEN
            MsgBox "Score must be between 0 and 100"
        END IF
        LOOP UNTIL tests(scores) >= 0 AND tests(scores) <= 100
    NEXT scores
average = (tests(1) + tests(2) + tests(3) + tests(4) + tests(5)) / 5
```

Use of White Space

White space is used to make code more readable by leaving blank lines between procedures.

This makes it easier to identify the control constructs in the code and to see where procedures start and finish.

Structured Listings

Indentation between subprograms helps to give the program listing a structure. It makes it easier to identify control constructs in the code, such as which sections of code are repeated and which instructions are selected for execution in an IFTHENELSE.....END IF constructs.

Also structured listings make it easier for programmer to identify where each subprogram starts and ends.

