National 5

Computing Science

Software Design and Development

Pupil Notes

Name:______________________________

CONTENTS

ANALYSIS STAGE

PROGRAM PURPOSE

The Analysis Stage is where we 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

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

Example:

Capturing Olympus

In the board game 'Capturing Olympus', six players work as a team to earn points. One point is earned if the six players score a combined total of more than 50 hits. An additional point is earned if the average number of hits is greater than or equal to 10.

Program analysis (purpose)
A program is required to determine the number of points earned by the team. The program will ask the user to enter the number of hits scored by each of the six players and store these values. When all six players' hits have been entered, the program will calculate the total and average number of hits. A message indicating the points earned is then displayed to the user.

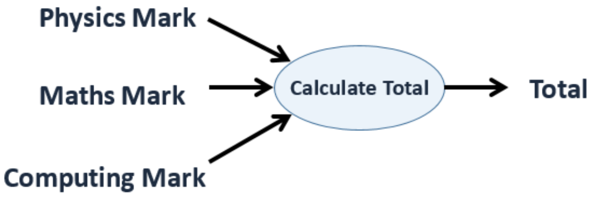
FUNCTIONAL REQUIREMENTS

The analysis stage help you to identify the program's functional requirements.

Functional requirements are a **list of tasks** that the **program must be able to do**.

The functional requirements include identifying the program:

- Inputs
- Processes
- Outputs



Inputs

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

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

Outputs

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.

Example:

Capturing Olympus

In the board game ‘Capturing Olympus’, six players work as a team to earn points. One point is earned if the six players score a combined total of more than 50 hits. An additional point is earned if the average number of hits is greater than or equal to 10.

Program analysis (purpose)
A program is required to determine the number of points earned by the team. The program will ask the user to enter the number of hits scored by each of the six players and store these values. When all six players’ hits have been entered, the program will calculate the total and average number of hits. A message indicating the points earned is then displayed to the user.

Inputs

- a valid number of hits scored by each of the six players

Processes

- calculate the total hits achieved by all six players
- calculate an average number of hits (total/6)
- determine if the six players have earned points

Outputs

- a message is displayed if one point has been earned
- a message is displayed if the additional point has been earned
- a message is displayed if no points have been earned

Assumptions

- the number of hits a single player can achieve is greater than or equal to 0 and less than or equal to 30
- the average should be displayed to two decimal places
- one point is earned if the total number of hits is greater than 50. An additional point is earned if the average number of hits is greater than or equal to 10

READING REVIEW 1

Having read pages 4 – 6, answer the questions below in preparation.

1. What is the purpose of the analysis stage?

____

____

____

2. Explain the following terms:

Input ____

____

Process____

____

Output____

____

3. Read the specification below

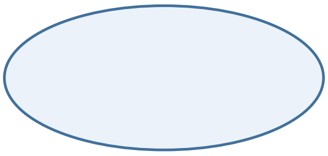
Draw arrows to show the inputs, processes and outputs on the diagram.

Problem Specification

A program is required to calculate the average number of goals scored in four recent football games.

Game 1 – 3 goals, Game 2 – 2 goals, Game 3 – 2 goals, Game 4 – 1 goal

The average will be found by adding the number of goals scored in each football game and then dividing by four. The program should display the average number of goals scored with a suitable message.



DESIGN

During the Design stage, a plan for how to solve the problem, as described in the *functional requirements*, is created.

DESIGN TECHNIQUES

A design technique is a method of representing the program and explaining how it should work.

Graphical Design Techniques

Graphical design techniques use diagrams made up of boxes or arrows to represent the steps of the program.

Examples are:

- Flow Chart
- Structure Diagram

An advantage of graphical design techniques is that they give a **visual representation** of the program structure and order of events. This gives a clear overview of the design and can make it easier to understand.

Text-Based Design Techniques

Text-based design techniques use numbered steps with written descriptions of each task carried out by the program.

Examples are:

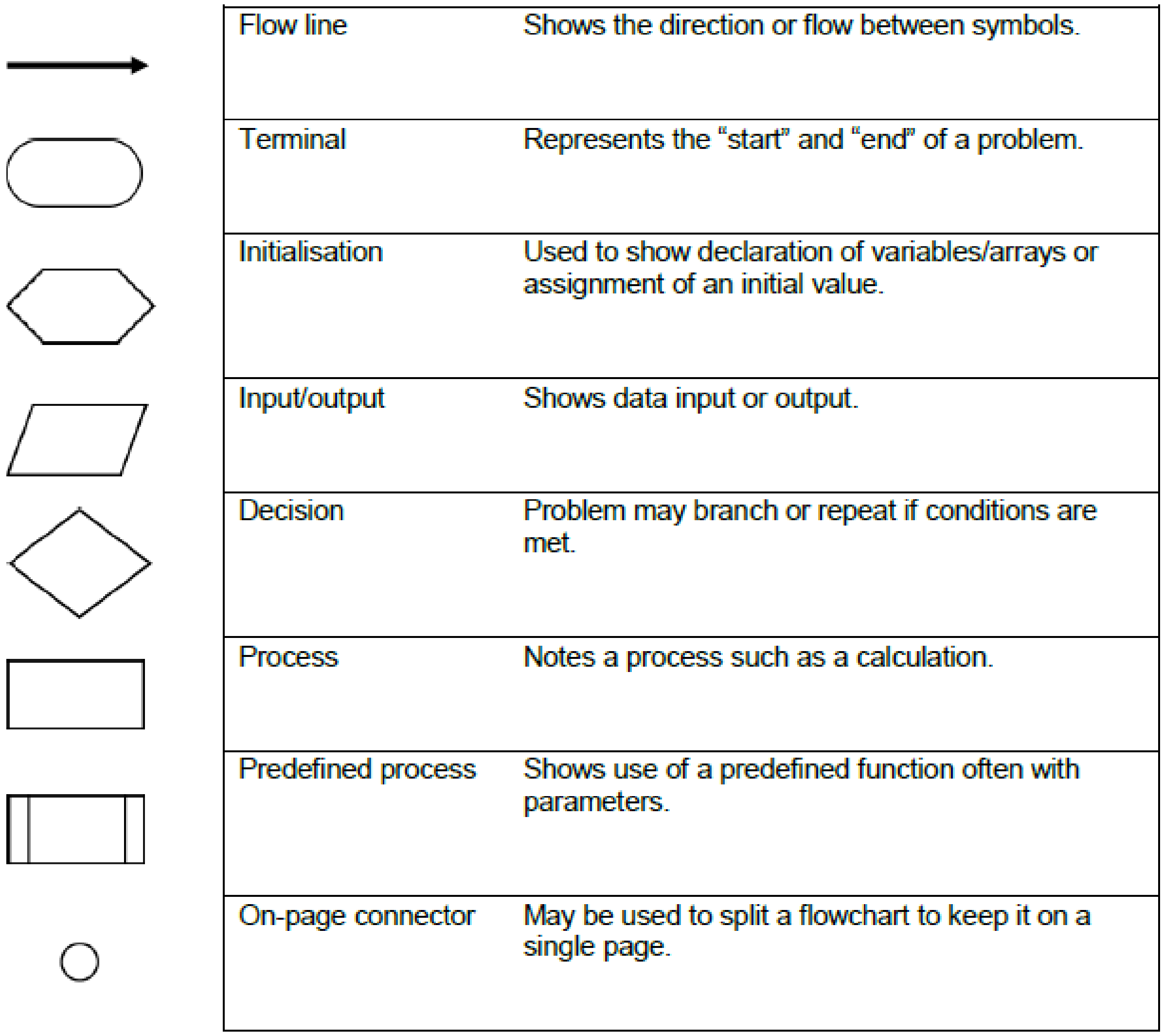
- Pseudocode

Flow Charts

A flowchart uses a variety of standard symbols with text to represent the order of events required to solve a problem.

The symbols in a flowchart can be equated to programming constructs such as assignment, selection and repetition.

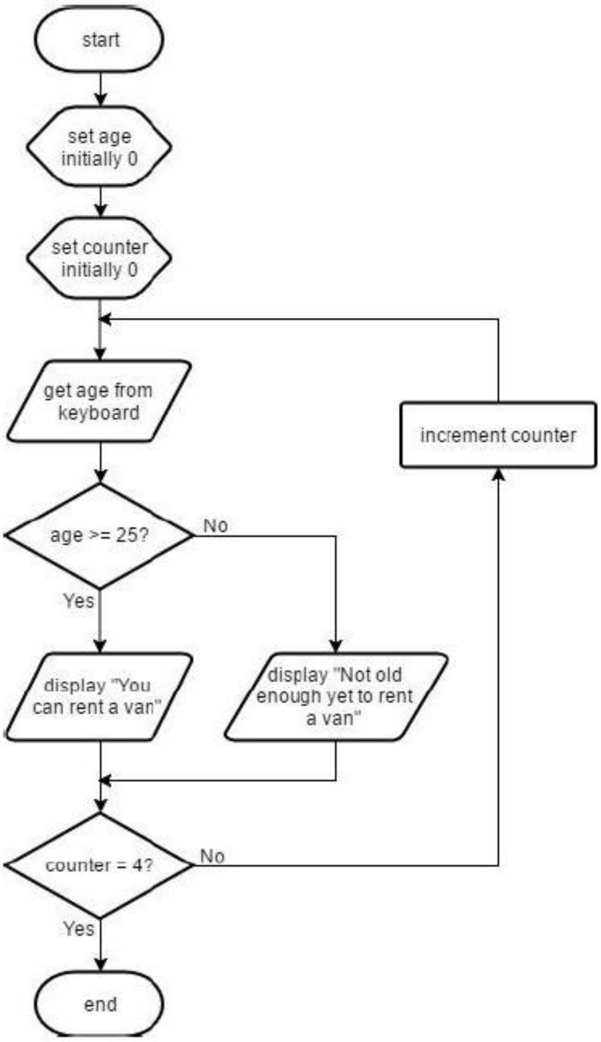
Flow Chart Symbols



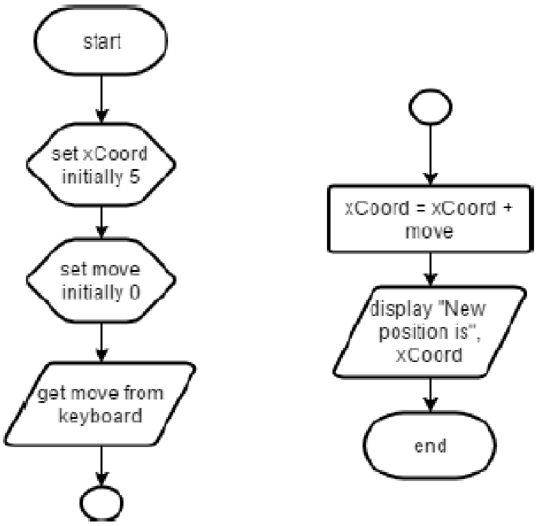
Flow line	Shows the direction or flow between symbols.
Terminal	Represents the “start” and “end” of a problem.
Initialisation	Used to show declaration of variables/arrays or assignment of an initial value.
Input/output	Shows data input or output.
Decision	Problem may branch or repeat if conditions are met.
Process	Notes a process such as a calculation.
Predefined process	Shows use of a predefined function often with parameters.
On-page connector	May be used to split a flowchart to keep it on a single page.

Flow Chart Examples

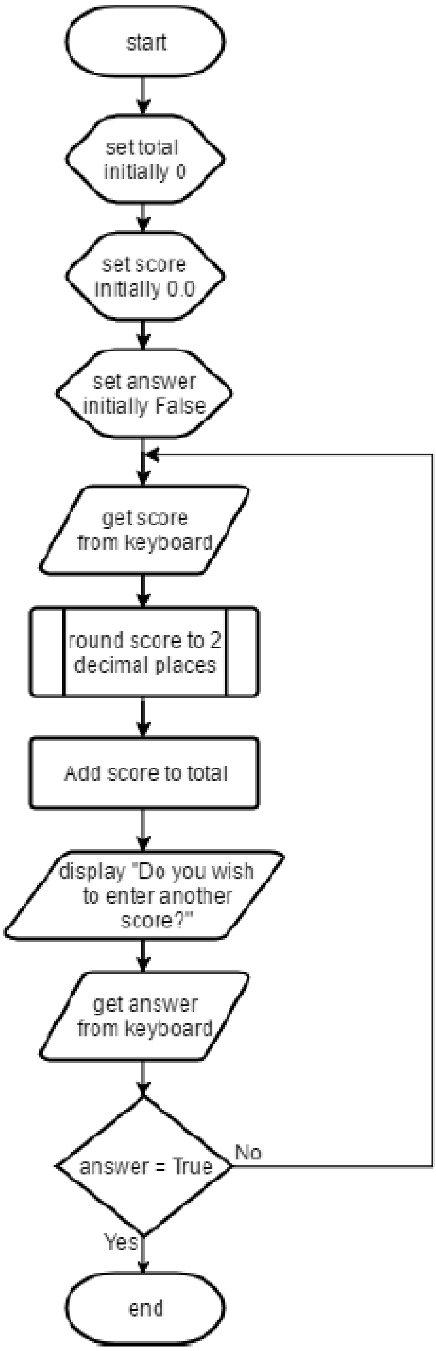
Example 1*:Fixed Loop*



Example 3*:On-page connector*

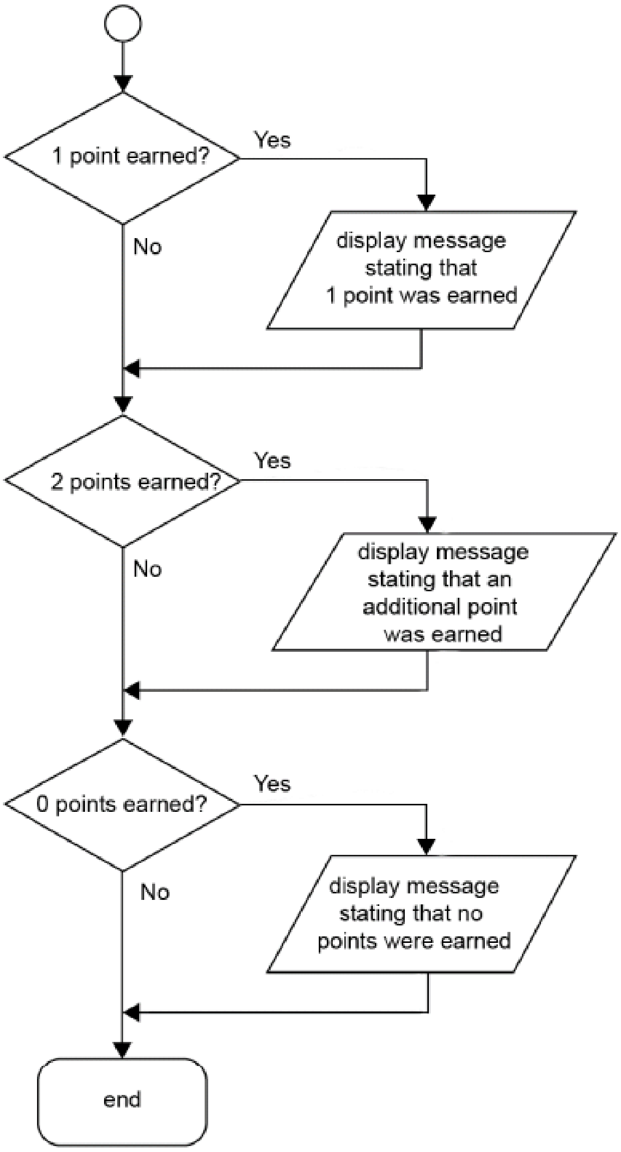


Example 2*:Conditional Loop*



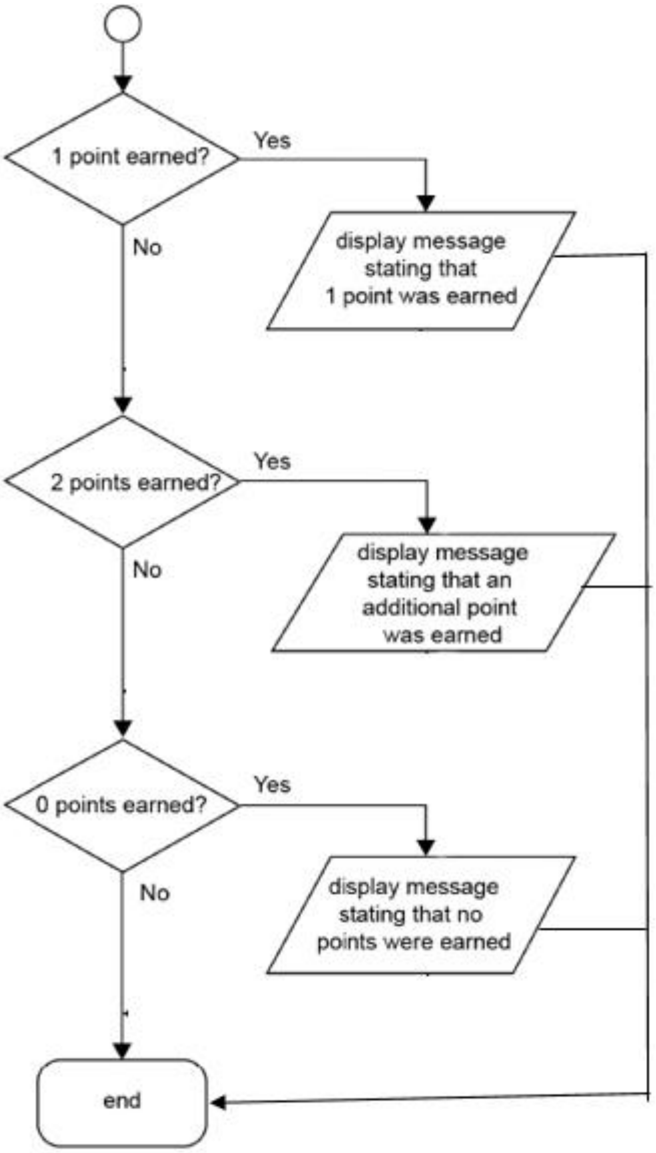
***Example 4a**: Separate Selection (IF)*

When separate IF statements are used, every condition will be checked, even after a yes is found


1 point earned?
Yes
No
display message stating that 1 point was earned
2 points earned?
Yes
No
display message stating that an additional point was earned
0 points earned?
Yes
No
display message stating that no points were earned
end

***Example 4b**: Nested Selection (IF)*

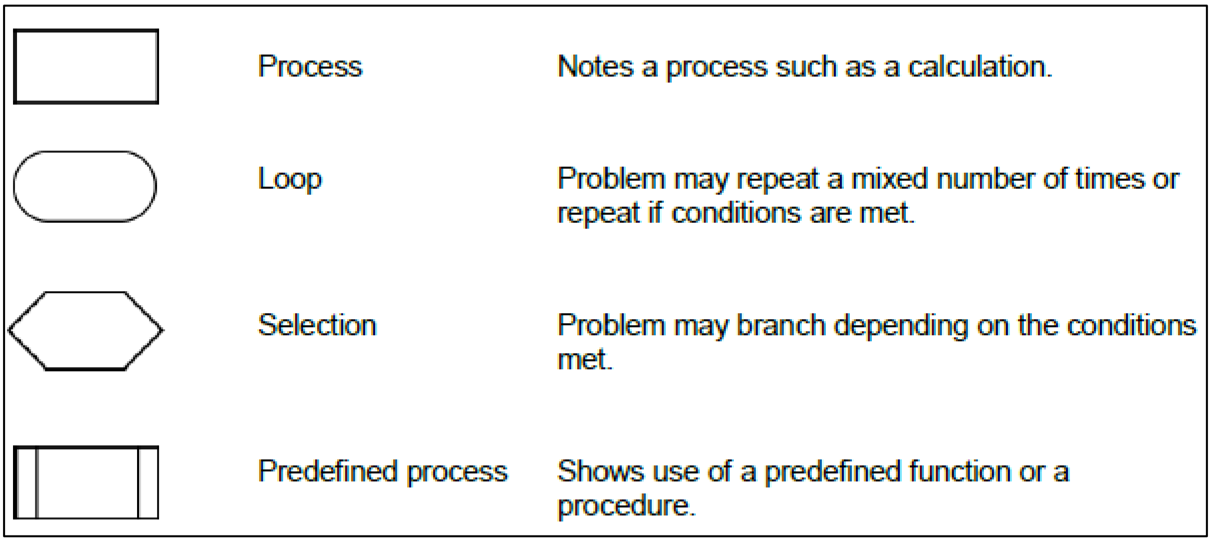
When nested IF statements are used, conditions will only be checked until the first yes is found.


1 point earned?
Yes
No
display message stating that 1 point was earned
2 points earned?
Yes
No
display message stating that an additional point was earned
0 points earned?
Yes
No
display message stating that no points were earned
end

Structure Diagrams

A structure diagram is a method of graphically representing the steps required to solve a problem. Structure diagrams must be read from the top down from left to right.

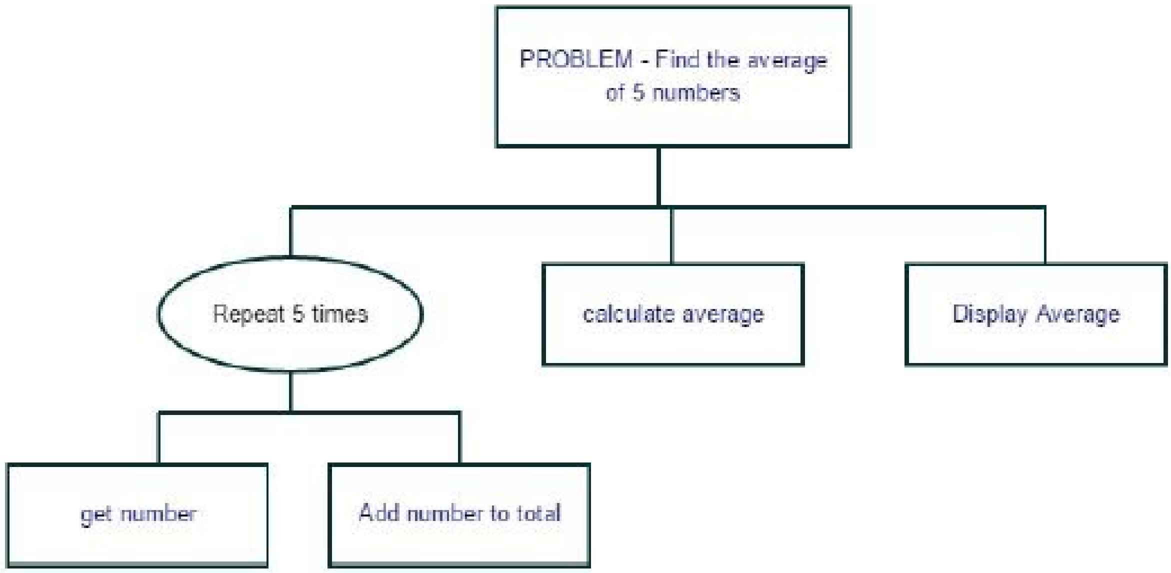
There are four types of notations used to represent the workings of the program:



▭	Process	Notes a process such as a calculation.
⬭	Loop	Problem may repeat a mixed number of times or repeat if conditions are met.
⬡	Selection	Problem may branch depending on the conditions met.
◫	Predefined process	Shows use of a predefined function or a procedure.

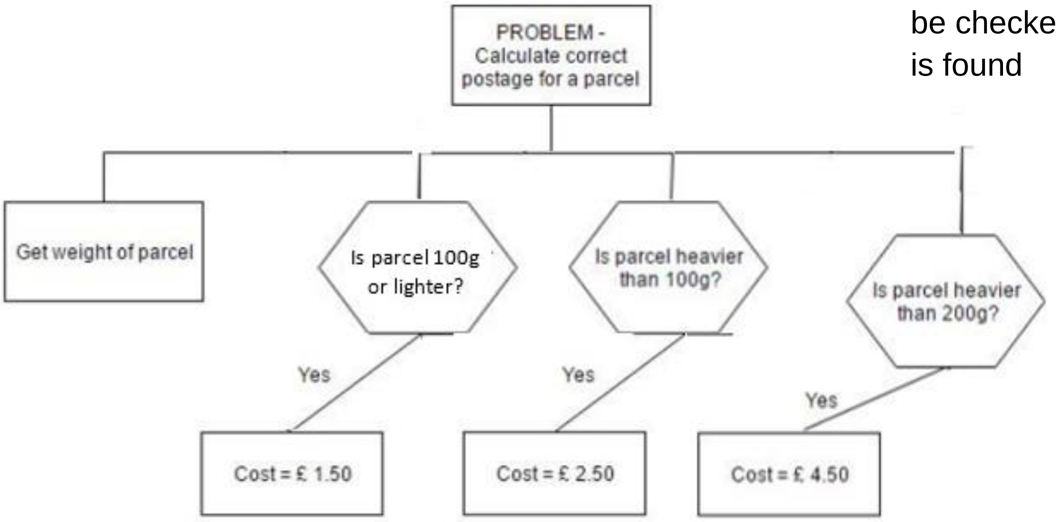
Structure Diagram Examples

Example 1*:Fixed Loop*



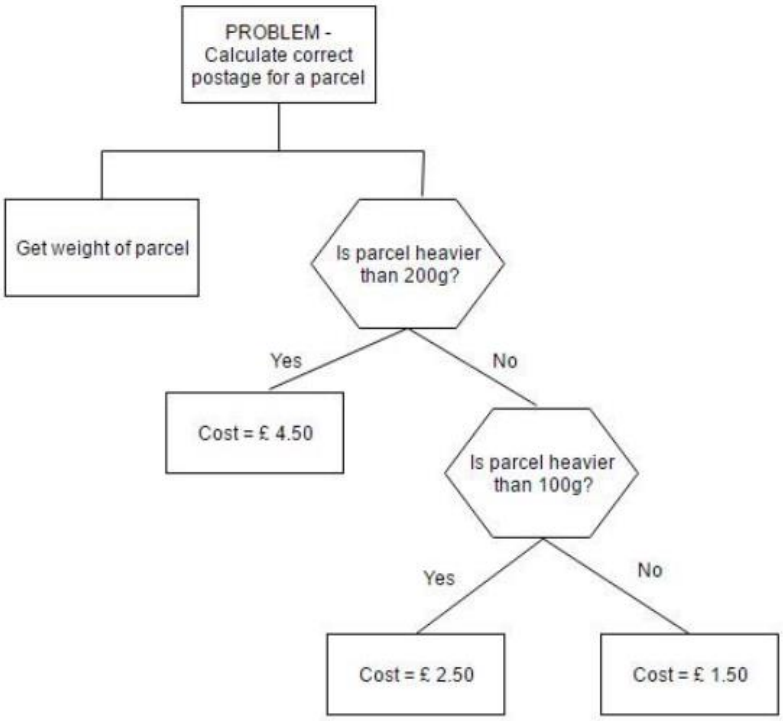
Example 2a*:* Separate selection (IF)

When separate IF statements are used, every condition will be checked, even after a yes is found


PROBLEM - Calculate correct postage for a parcel
Get weight of parcel
Is parcel 100g or lighter?
Is parcel heavier than 100g?
Is parcel heavier than 200g?
Yes
Yes
Yes
Cost = £ 1.50
Cost = £ 2.50
Cost = £ 4.50

Example 2b*:* Nested selection (IF)

When nested IF statements are used, conditions will only be checked until the first yes is found.


PROBLEM - Calculate correct postage for a parcel
Get weight of parcel
Is parcel heavier than 200g?
Yes
No
Cost = £ 4.50
Is parcel heavier than 100g?
Yes
No
Cost = £ 2.50
Cost = £ 1.50

Pseudocode

Pseudocode is a kind of structured English for describing algorithms and is intended for human reading.

It should not be written like program code and usually misses out things like declaring variables.

Pseudocode might look a bit like program code but it doesn’t have the same strict rules or syntax.

Example 1:

```
Algorithm

1 Initialise total length
2 Get valid number of tracks from user
3 Start fixed loop for each track
4    Get title and track length from user
5    Add track length to total
6 End fixed loop
7 Display track titles and track lengths
8 Display total length


Refinement
2.1 Start conditional loop
2.2 Get number of tracks from user
2.3 If number of tracks is not valid display error message
2.4 Repeat until the number of tracks entered is between 1
    and 20 inclusive

4.1 Get track title and store in names array
4.2 Get track length and store in length array

5.1 Add track length to total length

7.1 Start fixed loop for length of names array
7.2    Display “The name of track”, counter, “is”, track name
7.3    Display “The length of track”, counter, “is”, track
       length
7.4 End fixed loop

8.1 Display “The total length of the tracks is”, total length
```

Example 2:

```
Algorithm

1 Ask user to enter dimensions of a swimming pool in metres
2 Calculate volume of pool
3 Display message stating the volume of the pool


Refinement
1.1 Ask user to enter length of pool
1.2 Ask user to enter width of pool
1.3 Ask user to enter depth of pool

2.1 Volume is calculated as length * width * depth

3.1 Display “The volume of the pool is”, volume
```

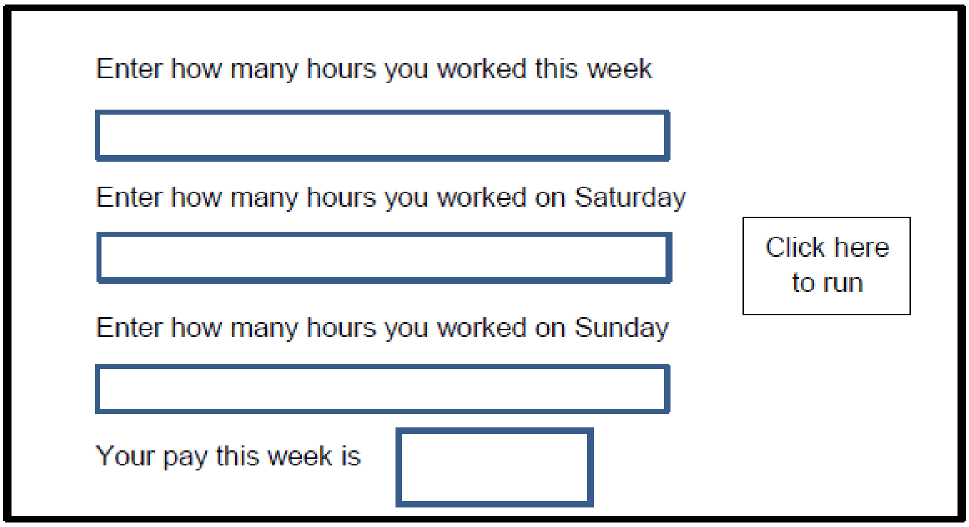
USER INTERFACE DESIGN

The user interface is the part of a computer program that is visible to the user.

The point of designing a user interface for software is to show what input and output is required, so that the programmer can implement it in their chosen code.

The type of user interface can depend on what the programming language is capable of achieving. The examples below would probably be sketched by the designer, rather than typed.

Example 1:
This interface design uses graphical text boxes and buttons



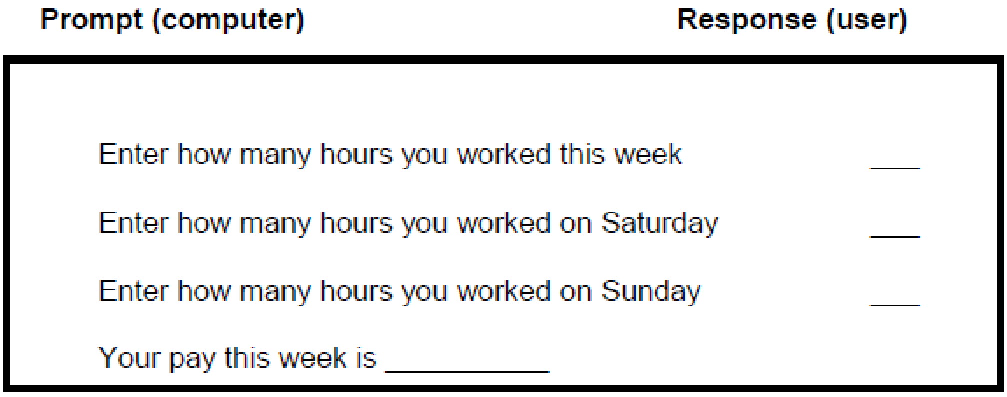
Enter how many hours you worked this week

Enter how many hours you worked on Saturday

Enter how many hours you worked on Sunday

Your pay this week is

Example 2:
This is a simple text-based used interface.



Prompt (computer) **Response (user)**

Enter how many hours you worked this week ____

Enter how many hours you worked on Saturday ____

Enter how many hours you worked on Sunday ____

Your pay this week is ___________

READING REVIEW 2

Having read pages 8 – 16, answer the questions below in preparation.

1. Explain why it is important to spend time on the design stage during the software development process.

2. Name a graphical design technique.

3. Name a text based design technique.

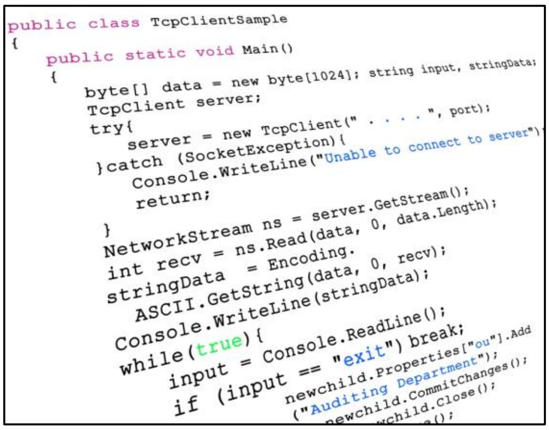
4. Explain the advantages of a graphical design technique over a text based design technique.

5. Explain the purpose of a wireframe diagram.

IMPLEMENTATION

The implementation stage is where the programming actually takes place.

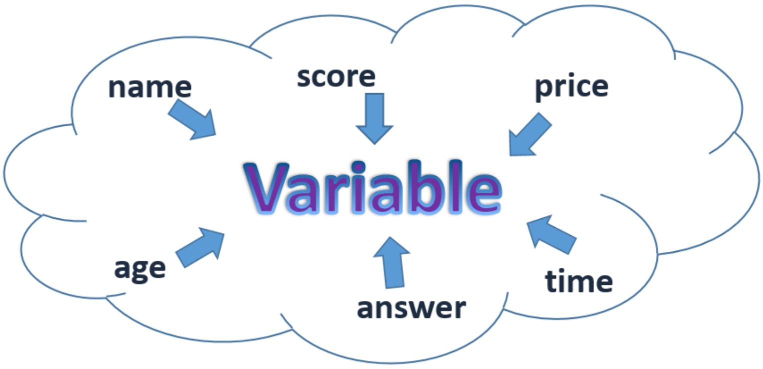
The user interface for the program is created and design documentation is used and converted into **high level language** instructions.



```
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();
                ();
```

VARIABLES AND DATA TYPES

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.

Declaring

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

DECLARE *score* AS INTEGER

or **DECLARE *score* INITIALLY 0**

Meaningful identifier

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****.*

Data Types

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

This decides whether the variable can store text or numbers for example.

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

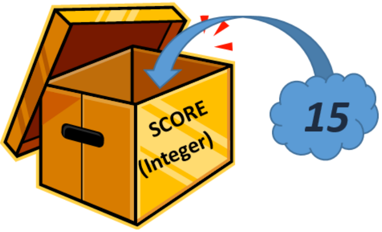
There are five main data types that variables can be used to store:

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 or False	TRUE / FALSE

Assigning Values to variables

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

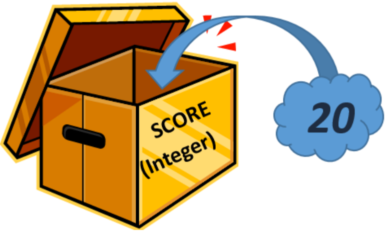
```
SET score TO 15
```



The score variable now contains the value 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
```



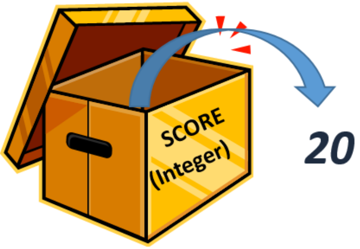
The score variable now contains the value 20

(15 has been deleted)

Outputting variables

To display the contents of a variable (**output**), 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.

Initialising Variables

It is good practice to set an initial value for variables at the start of a program, even if this value is zero.

DECLARE *score* INITIALLY 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 (String)***
- ***Heat1_time (Real)***
- ***Heat2_time (Real)***
- ***Best_time (Real)***

COMPUTATIONAL CONSTRUCTS

Computational constructs are used by programs to allow them to control data, carry out calculations, make decisions and perform repetitive tasks.

There are a number of constructs that you have to know for this course

Construct	**Example** (***SQA Reference Language***)

Expressions to Assign Values	**SET** total TO 0
	SET answer TO “Computing”
	RECEIVE username FROM (INTEGER) KEYBOARD

Expressions to Return Values using Arithmetic Operations	**SET** total TO num1 + num2
	SET average TO total / 7

Expressions to Concatenate Strings	SET answer TO “Computing” **&** “Science”
	SET final TO answer **&** “is best”
	SEND “Your answer is ” **&** answer

Selection	**IF** total = 5 THEN
	        SET answer TO “Correct”
	ELSE
	        SET answer TO “Wrong”
	END IF

Logical Operators	total > 5 **AND** total <10 total <= 5 **OR** total >=10 **NOT**(total > 10)

Iteration and Repetition	**REPEAT**
	        RECEIVE total FROM (INTEGER) KEYBOARD
	UNTIL total > 5 AND total <10
	
	REPEAT index FROM 1 TO 10
	        SEND “Hi there” TO DISPLAY
	END REPEAT

Pre-Defined Functions	SET num TO **ROUND**(average, 2)
	SET num TO **RANDOM** (6)
	SET num TO **LEN** (answer)

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 be a line of code containing an equals (=) symbol.

Expressions to Assign Values

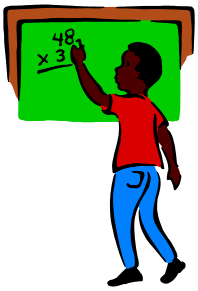
Expressions are used to assign a value to a variable. This could be a newly created variable that is being **initialised.**

```
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 num1 + 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 in some languages.

Note: Other programming languages may use different symbols for concatenation.

Output Variables with a Message

Concatenation allows us to add a meaningful message when outputting variables. This makes a program more user-friendly.

Examples:

Three variables contain the values shown:

Variable	answer	firstName	surname
Contents	43	Jane	Jones

```
SEND “Answer is: ” & answer TO DISPLAY
```

Output would be:
Answer is 43

```
SEND “Hello ” & firstName & surname TO DISPLAY
```

Output would be:
Hello JaneJones

Notice in the second example, there is no space between the first name and surname. This is because a space was not concatenated between them in the SEND line.

To fix this problem, the following change should be made.

```
SEND “Hello ” & firstName & “ ” & surname TO DISPLAY
```

Output would be:
Hello Jane Jones

Appending Strings

Concatenation can also be used to allow a variable to add characters to itself.

Example:

In this example, firstname starts as an empty variable. Each time the expression adds a new letter to firstname building.

letter	Expression	firstname
F	**SET *firstName* TO *firstName* & *letter***	F
r	**SET *firstName* TO *firstName* & *letter***	Fr
e	**SET *firstName* TO *firstName* & *letter***	Fre
d	**SET *firstName* TO *firstName* & *letter***	Fred

We cannot simple say: **SET *firstName* TO *letter*** otherwise the previous letters would be overwritten each time.

By using **SET *firstName* TO *firstName* & *letter*** the each letter is added to the end of current contents of the firstname variable.

READING REVIEW 2

Having read pages 17 – 27, answer the questions below in preparation.

1. What is a variable?

____

____

____

____

2. What data type would be used to store the following information:

Hello ____________________

13.99 ____________________

KA15 6DX ____________________

10001 ____________________

True ____________________

3. Explain the following terms:

Declaring

____

Initialising

____

Assigning

____

Concatenation

____

4. Complete the table below:

Operation	**Symbol**
Subtraction	
	+
Multiplication	
	^
Division	

Selection

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

IF Statement

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
2.     Go to school
3. END IF
4. …
```

Line 1 is our condition

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

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.

IF – ELSE (Separate IF)

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
2.       Go to school
3. ELSE
4.       Go shopping
5. END IF
6. …
```

Line 1 is our condition

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

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

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

IF - ELSE IF – ELSE (Nested IF)

We could decide to do a different activity depending on whether it is Saturday or a Sunday.

Now, our morning rule might be:

```
IF today is a weekday THEN
        go to school
BUT IF today is Saturday THEN
        go to the shops
OR ELSE
        stay at home
```

We use the **ELSE IF** command to indicate an alternative option if the first condition is false. Else is then used if the second condition is also false.

```
1. IF today = "weekday"  THEN
2.        Go to school
3. ELSE IF today = "Saturday"  THEN
4.        Go shopping
5. ELSE
6.        Stay at home
7. END IF
8. ...
```

Line 1 is our first condition

Line 3 is only carried out if line 1 is false

Line 4 is **only** carried out if the lines 1 and 3 are **false**

Notice, after a true condition is found, all other conditions are bypassed.

Simple conditions can use the following operations:

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

Nested IF v Separate IF

It is more efficient to use nested IF statements instead of separate IFs, especially when there are a lot of options.

```
1. IF score >= 3 THEN
2.     SEND "Excellent" TO DISPLAY
3. ELSE IF score = 2 THEN
4.     SEND "Good" TO DISPLAY
5. ELSE IF score = 1 THEN
6.     SEND "Nice try" TO DISPLAY
7. ELSE
8.         SEND "Try again" TO DISPLAY
9. END IF
```

If the score is 3 or more, line 2 is executed and then lines 3 to 8 are skipped.

```
1.  IF score >= 3 THEN
2.      SEND "Excellent" TO DISPLAY
3.  END IF
4.  IF score = 2 THEN
5.          SEND "Good" TO DISPLAY
6.  END IF
7.  IF score = 1 THEN
8.      SEND "Nice try" TO DISPLAY
9.  END IF
10. IF score < 1 THEN
11.         SEND "Try again" TO DISPLAY
12. END IF
```

If the score is 3 or more, line 2 is executed and then the condition at line 4 is checked.

In this example, if the score is 3 then it obviously cannot also be 2 or 1.

Nested IF statements are more efficient as they prevent code being executed when it is unnecessary.

Separate IF statements will continue to check conditions even if it is impossible for them to be true.

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 repeating instructions inside it.

```
1.  FOR loops FROM 1 TO 5 DO
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.

```
REPEAT
        Walk UP stairs
        Walk DOWN stairs
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**.

```
REPEAT
        Walk UP stairs
        Walk DOWN stairs
UNTIL time = 12:30
```

These instructions will be repeated

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.

```
WHILE time <> 12:30 DO
    Walk UP stairs
    Walk DOWN stairs
END WHILE
```

These instructions will be repeated

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

Complex Conditions

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

```
1. IF today = "Saturday" OR today = "Sunday" THEN
2.     Go to the beach
3. ELSE
```

Line 1 contains a complex condition with two things to check:
Is it Saturday or is it Sunday

Notice that for each part of the complex condition, we must specify the variable.

We **cannot** write, **IF today = "Saturday" OR "Sunday"**.

READING REVIEW 4

Having read pages 29 – 36, answer the questions below in preparation.

1. What is a selection construct?

____

____

2. A computer program is used to check if a contestant will progress to the next round of auditions. A program is required to display a congratulations message if the contestant scores 20 or more points in their audition. If they score less than 20 an unsuccessful message is displayed. Using pseudocode code or a programming language show how this code would be implemented.

3. State the **type of loop** shown in the code below:

```
REPEAT
        RECEIVE pupilAge FROM (INTEGER) KEYBOARD
UNTIL pupilAge <= 18
```

____

4. State the type of loop shown in the code below:

```
FOR loops = 1 to 5 DO
        price = Inputbox(“Enter price”)
        Total = total + price
END FOR
```

____

5. Explain the difference between a fixed loop and a conditional loop.

____

____

____

Logical Operators

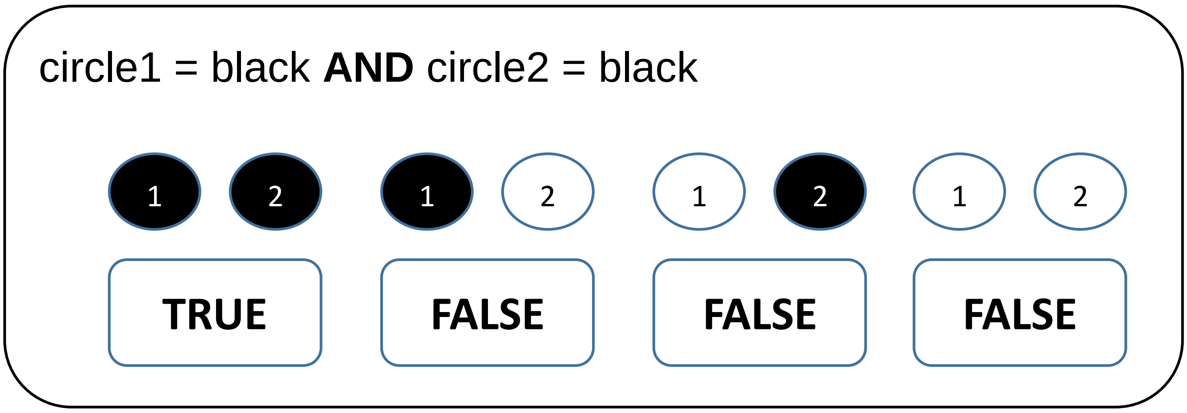
Logical Operations are used to create **complex conditions**. Complex conditions are conditions that have **two or more** parts to them.

The main logical operators are:

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

AND Logical Operator

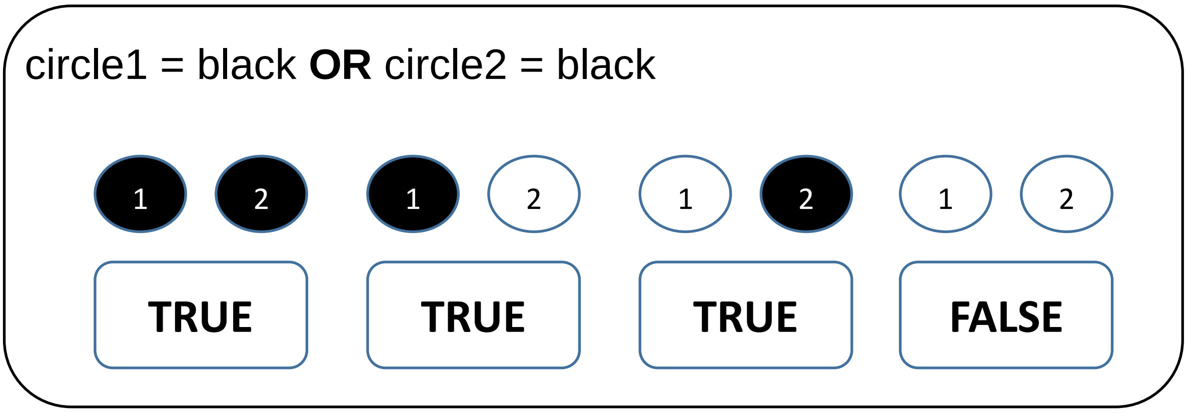
AND checks that **both** parts of a condition is true



circle1 = black **AND** circle2 = black

OR Logical Operator

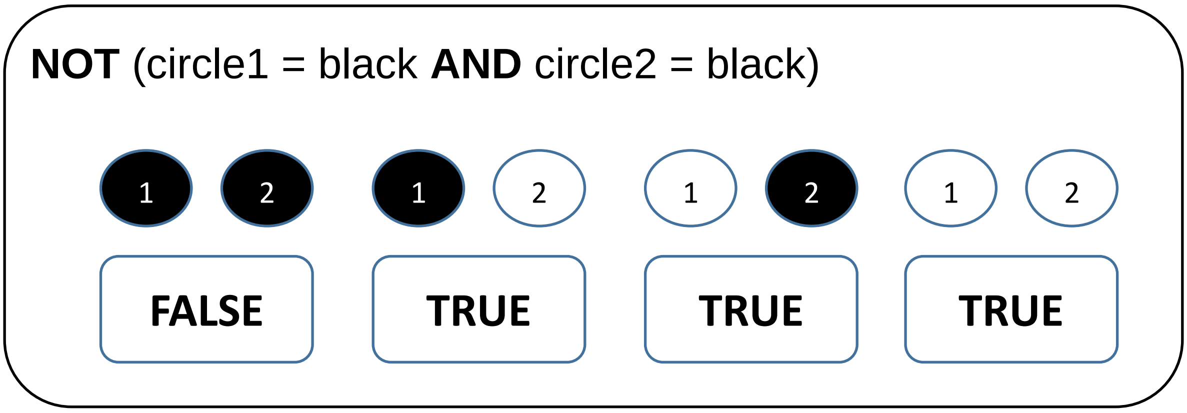
OR checks that **either** part of a condition is true



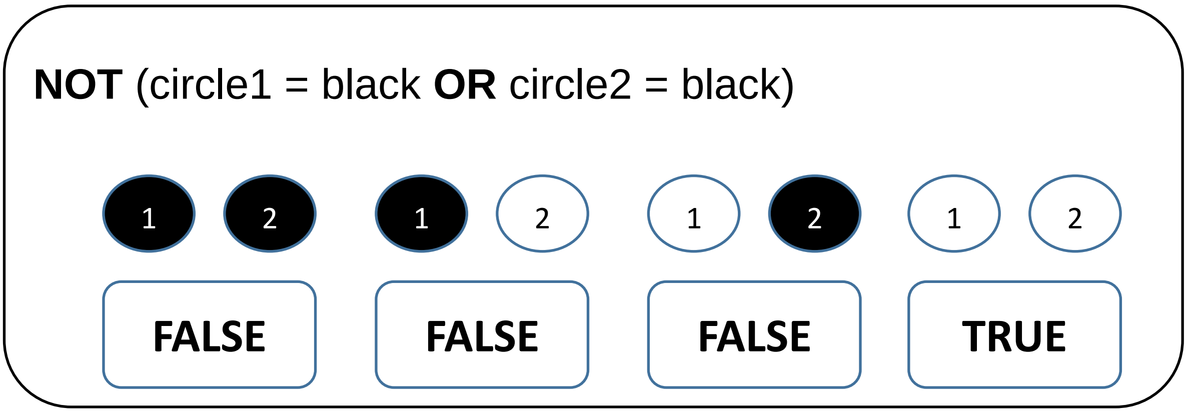
circle1 = black **OR** circle2 = black

Not Logical Operator

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 ***condition1*** **AND** ***condition 2***

IF ***condition1*** **OR** ***condition 2***

IF NOT (*condition1* OR *condition 2*)

UNTIL ***condition1*** **AND** ***condition 2***

DO WHILE ***condition1*** **OR** ***condition 2***

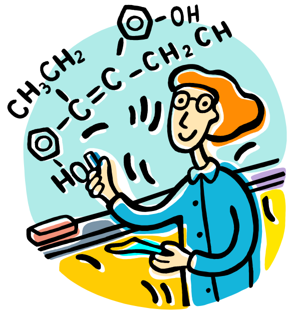
UNTIL NOT (*condition1* OR *condition 2*)

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”
```

Pre-defined 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.

Format of a function

answerVariable **=** ***functionName*** (parameter1, parameter2, …)

This can be any variable and is used to store the answer returned by the function. The variable data type must match the type of value returned by the function.	This is the name of the pre-defined to be used. *See the names of functions below.*	Parameters are the inputs required by the function. A function can have none, one or more parameters – it depends on the function.

Types of Function

There are three functions you must be able to use in this course.

Function	**Purpose**	**Returned Data Type**	**Parameters**
RANDOM	Returns a random number in a specified range	Integer	*none*
LENGTH	Returns the number of characters in a string	Integer	• a character string
ROUND	Returns a rounded real number to a specified number of decimal places	Integer	• A real number • Decimal places required

Function Examples

Random

Use a function to generate a random number between 1 and 7

```
chosenNumber = RANDOM( ) * 7
```

Length

Use a function to return the number of characters in a string stored in a variable called sportsTeam.

```
teamLength = LENGTH(sportsTeam)
```

Round

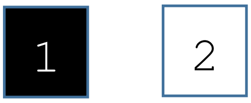
Use a function to round a real number stored in a variable call average to 2 decimal places.

```
roundedAvg = ROUND(average, 2)
```

READING REVIEW 5

Having read pages 38 – 42, answer the questions below in preparation.

1. Two squares, one coloured black and the other white are to be compared.



For each complex condition below, decide whether the result would be TRUE or FALSE.

(a) Square1 = Black AND Square2 = White ______________________

(b) Square1 = White OR Square2 = White ______________________

(c) Square1 = Black AND Square2 = Black ______________________

(d) NOT (Square1 = White) OR Square2 = Black ______________________

2. State the purpose of each of the following pre-defined functions:

Random:____

____

Length:____

____

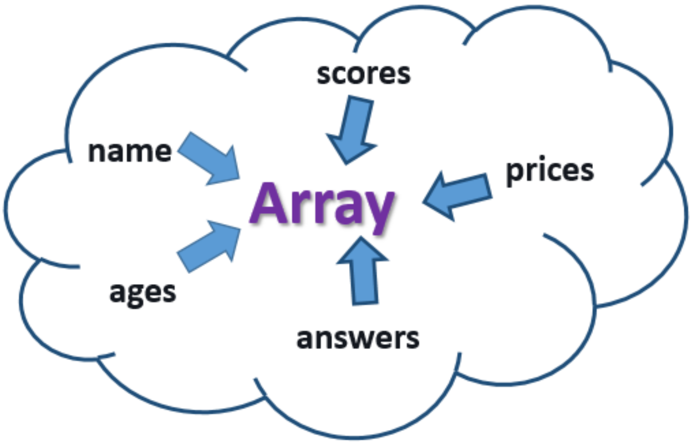
Round:____

____

DATA STRUCTURES

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**.



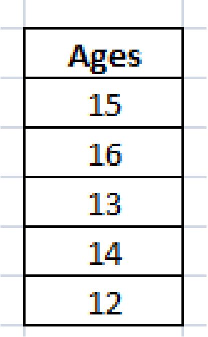
Imagine an array as being like a **list** of items that must all be on the same subject.

Declaring arrays (meaningful identifier, data type and size)

Creating a new array is called **declaring.** When declaring an array, there are three things that must be specified.

1. 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**.*



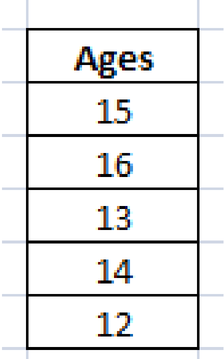
Ages
15
16
13
14
12

2. 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)



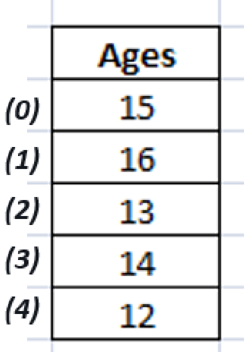
Ages
15
16
13
14
12

3. 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

```
DECLARE Ages  AS ARRAY OF INTEGER INITIALLY [ ]
```

or

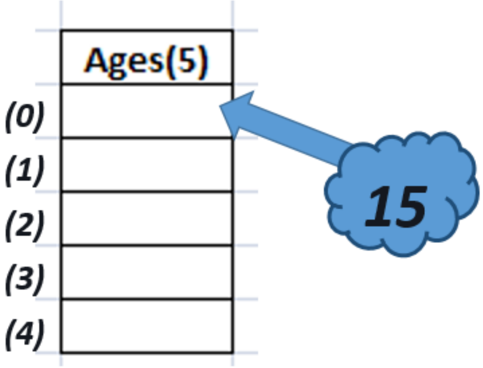
```
DECLARE Ages  INITIALLY [15,16,13,14,12 ]
```

Assigning Values to Arrays

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
```

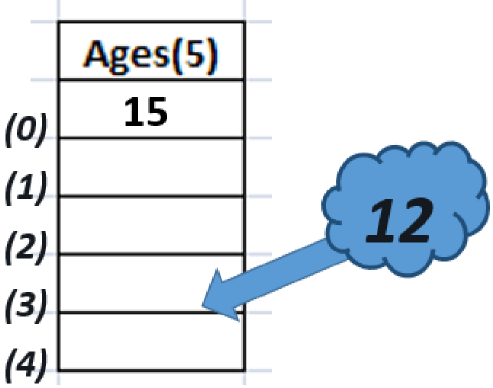
Position 0 of the age array now contains the value 15.



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

```
SET Ages[3]TO 12
```

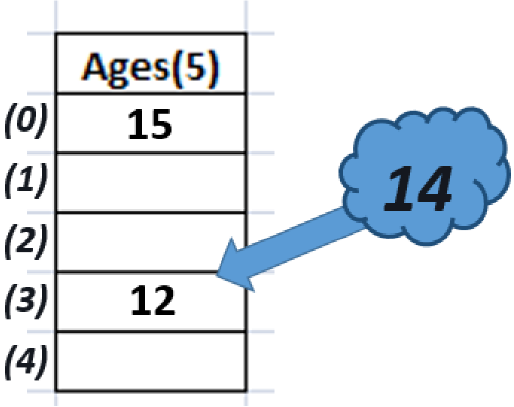
Position 3 of the age array now contains the value 12



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

```
SET Ages[3]TO 14
```

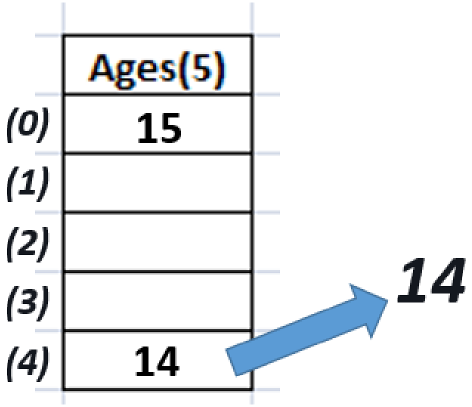
Position 3 of the age array now contains the value 14



Outputting Arrays

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

```
SEND Ages[3]TO DISPLAY
```



	Ages(5)
(0)	15
(1)	
(2)	
(3)	
(4)	14

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

Initialising Arrays

An array can be initialised with values in order that it can be used without the user having to enter data first.

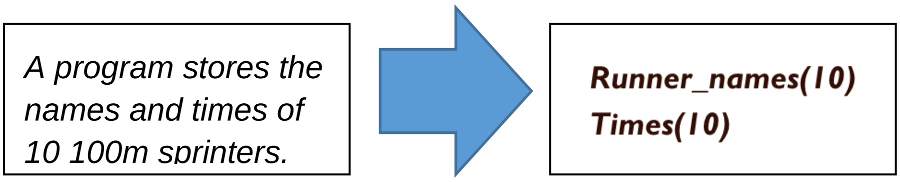
```
DECLARE Ages INITIALLY [15,16,13,14,12]
```

Ages
15
16
13
14
12

Score is now set back to 0

When to use Arrays

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



A program stores the names and times of 10 100m sprinters.

Runner_names(10)
Times(10)

READING REVIEW 6

Having read pages 43 – 48, answer the questions below in preparation.

1. Explain why a 1D array is useful in computer programs.

    ____

    ____

2. When declaring an array, what three features of the array must be specified?

    ____

    ____

    ____

STANDARD ALGORITHMS

A standard algorithm is a step-by-step way to solve a particular problem.

These standard steps can be used in many programs, with minor changes, where the same problem needs to be solved.

There are three standard algorithms you have to learn in this course.

- Running Total within a Loop
- Input Validation
- Traversing a 1D Array

Running Total within a Loop

This algorithm is used to add up a series of values either entered at the keyboard or contained in an array.

It is far more efficient to use a loop for this than adding the values as one long calculation.

If you had 3 values to enter, you could just write.

SET total TO value1 + value2 + value3

What if you have 103 values? Or 2000 values?

Using the method above, if you were asked to add up five values entered at the keyboard, you would have to write a lot of code.

The algorithm

```
DECLARE total INITIALLY 0

FOR index FROM 1 TO 5 DO
    RECEIVE newValue FROM KEYBOARD
    SET total TO total + newValue
END FOR
```

Total is a variable that is repeatedly added to with the newValue entered at the keyboard.

How it works

The trace table below demonstrates how the variables, newValue and total, are updated as the algorithm repeats.

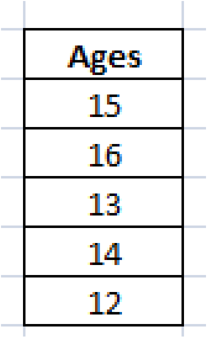
Line	number	total	*explanation*
1		0	total initialised to 0
3	5		5 typed in at the keyboard
4		5	total (0) + number (2) is 5
3	2		2 typed in at the keyboard
4		7	total (5) + number (2) is 7
3	3		3 typed in at the keyboard
4		10	total (7) + number (3) is 10
3	1		1 typed in at the keyboard
4		11	total (10) + number (1) is 11
3	2		2 typed in at the keyboard
4		13	total (11) + number (2) is 13

Running Total with Arrays

When an array contains a list of values to be added, the same algorithm can be used.

To add up all of the ages in this array, the Running Total algorithm would look like this:

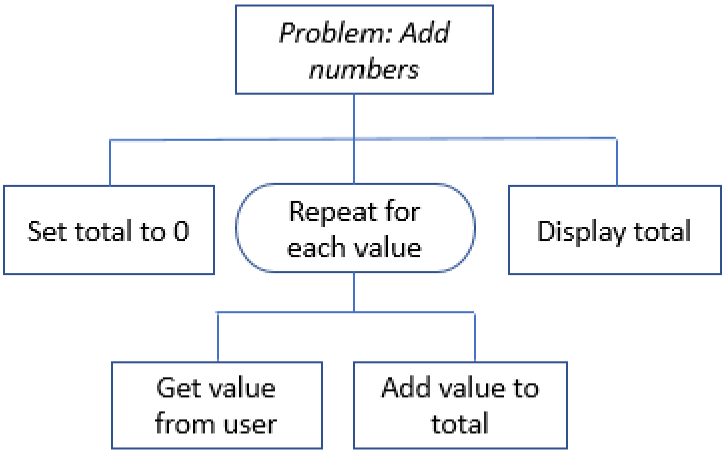
```
FOR index FROM  1 TO 50 DO
        total = total + ages(index)
END FOR
```



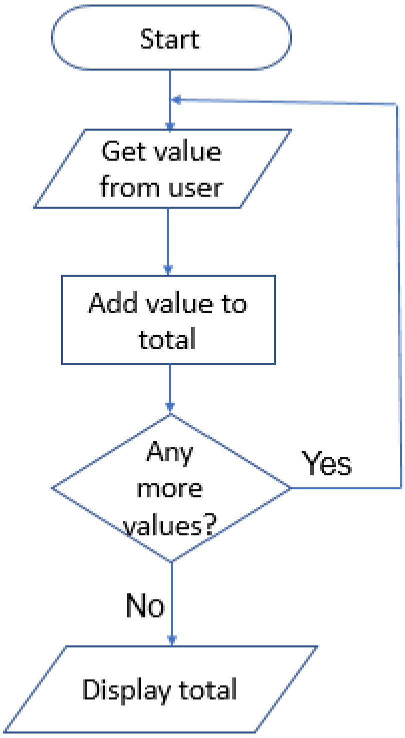
Ages
15
16
13
14
12

Running Total (Using Design Techniques)

Structure Diagram



Flow Chart



Pseudocode

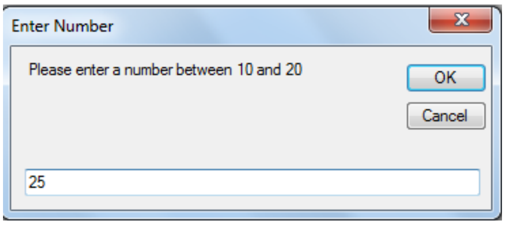
```
1.1 set total to 0
1.2     start loop for each value
1.3             get value from user
1.4             add value to total
1.5     end loop
1.6 display total
```

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.

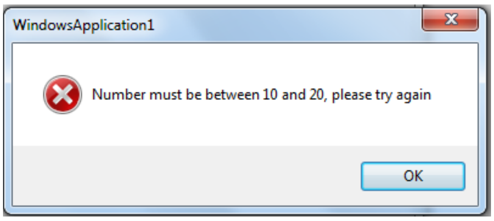


Enter Number

Please enter a number between 10 and 20

25

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.



WindowsApplication1

Number must be between 10 and 20, please try again

The steps of the Input Validation algorithm are **always** the same, regardless of the program you include it in.

The algorithm

```
1. REPEAT
2.       RECEIVE newValue FROM KEYBOARD
3.       IF newValue < 0  OR newValue > 10 THEN
4.             SEND  "Value is invalid"   TO DISPLAY
5.       END IF
6. UNTIL newValue >= 0  AND newValue <= 10
```

Line 1 – a **conditional loop** is started

Line 2 – a value is entered by the user

Line 3 – a **complex condition** is used to check if the entered value is valid

Line 4 – if the condition at line 3 is true, an error message is displayed

Line 6 – if the complex condition in the loop is false, the loop repeats from line 1

– if the complex condition in the loop is true, the loop terminates.

Examples

Example: Collecting month number from user

```
REPEAT
    RECEIVE month FROM KEYBOARD
    IF month < 1 OR month > 12 THEN
        SEND “Month is invalid” TO DISPLAY
    END IF
UNTIL month >= 1 AND month <= 12
```

Notice that the only changes in this example is the name of the variable and the range (1 to 12).

Example: Collecting age of secondary school pupils

```
REPEAT
    RECEIVE age FROM KEYBOARD
    IF age < 11 OR age > 18 THEN
        SEND “Month is invalid” TO DISPLAY
    END IF
UNTIL age >= 11 AND age <= 18
```

Again, all that has changed is the name of the variable and the range (11 to 18).

Example: Asking the user to enter only yes or no

```
REPEAT
    RECEIVE response FROM KEYBOARD
    IF response <> “yes” AND response <> “No” THEN
        SEND “Response is invalid” TO DISPLAY
    END IF
UNTIL response = “yes” OR response = “No”
```

This time, we are validating text entry instead of numbers so the conditions are slightly different.

Explaining Input Validation Code

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

```
1.  REPEAT
2.      RECEIVE month FROM KEYBOARD
3.      IF month < 1 OR month > 12 THEN
4.          SEND "Month is invalid" TO DISPLAY
5.      END IF
6.  UNTIL month >= 1 AND month <= 12
```

- 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

```
1.  REPEAT
2.      RECEIVE age FROM KEYBOARD
3.      IF age < 11 OR age > 18 THEN
4.          SEND "Month is invalid" TO DISPLAY
5.      END IF
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**

Input Validation with Arrays

When using a loop to input data into an array from the keyboard, each item should also be validated.

To validate each input in an array, a fixed loop must be added which repeats for the size of the array.

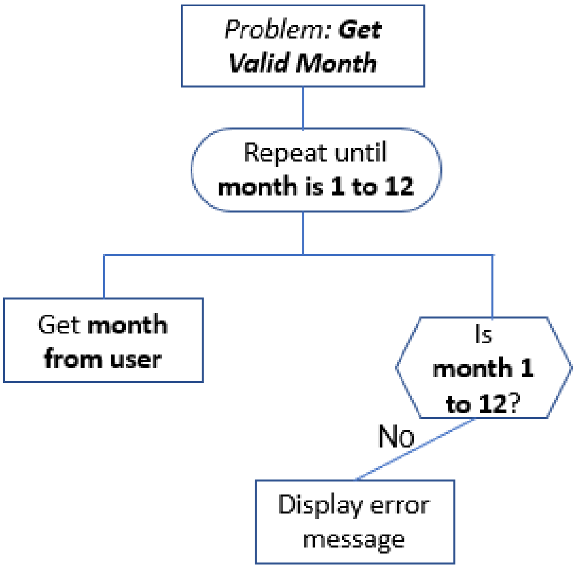
```
FOR index FROM 1 TO 5 DO
    REPEAT
        RECEIVE age(index) FROM KEYBOARD
        IF age(index) < 11 OR age(index) > 18 THEN
            SEND "Month is invalid" TO DISPLAY
        END IF
    UNTIL age(index) >= 11 AND age(index) <= 18
END FOR
```

Fixed loop repeats for each age to be entered

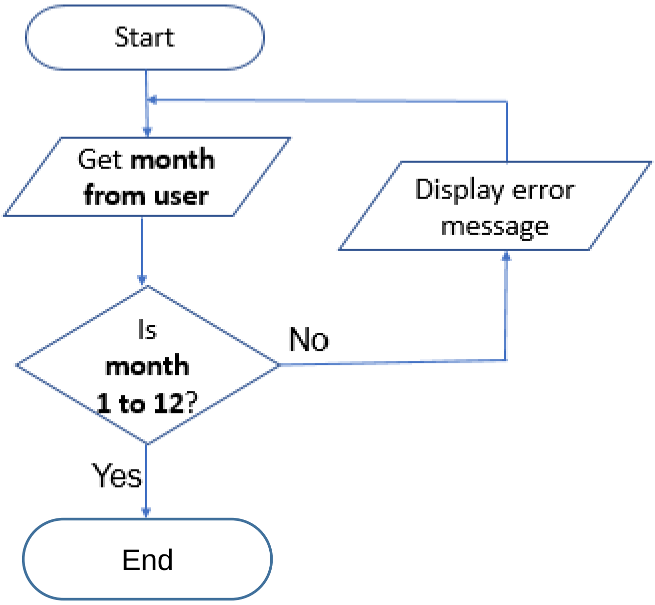
Every item inputted is validated using Input Validation

Input Validation (Using Design Techniques)

Structure Diagram



Flow Chart



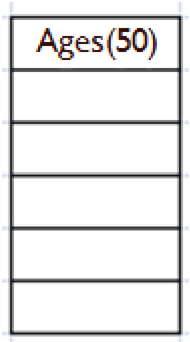
Pseudocode

```
1.1 start conditional loop
1.2     get month from user
1.3     if month is not between 1 and 12 then
1.4             display error message
1.5     end if
1.6  repeat until month is between 1 and 12
```

Traversing a 1D array

In a large array it may not be practical to input 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
```



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

Using a loop makes life much easier.

FOR Loop	FOR EACH Loop
FOR *index* FROM 1 TO 50 DO RECEIVE *age*(*index*) FROM KEYBOARD END FOR	FOR EACH *age* FROM *ages* RECEIVE *age* FROM KEYBOARD END FOREACH

index is used here to keep track on the current position in the array. Index will start at 1 and increment during each loop until it gets to 50.

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

READING REVIEW 7

Having read pages 50 – 58, answer the questions below in preparation.

1. State the purpose of a Running Total algorithm.

2. Write the steps of the running total algorithm.

3. State the purpose of the Input Validation algorithm.

4. Write the steps of the Input Validation algorithm.

5. Which programming construct is used to traverse a ID array **and why**?

TESTING

It is important to test your program to make sure that it:

- Runs without crashing
- Produces the correct output.

ERROR TYPES

There are three types of error that can occur in a program.

- Syntax Error
- Execution Error
- Logic Error

Syntax Error

Syntax is the rules of how a programming language must be written.

If code is written that **breaks the programming language rules** then a syntax error occurs.

The program **will not run at all** if there is a syntax error in the code.

Examples

Example 1	*Example 2*	*Example 3*

FOR index 1 TO 10
NEXT index
	DIM first IS STRING	IF first = 10 OR 20
		END IF

There is an equals sign missing after the word index. FOR **index = 1** TO 10 NEXT index	The word IS should be **AS** DIM first AS STRING	A complex condition must contain complete conditions on each side of the logical operator (OR). IF first = 10 OR **first =** 20

Execution Error

An execution error occurs when the program tries to do something impossible during while the program is running.

An execution error will **cause the program to crash** mid-way through. Usually an error message will appear explaining the problem.

Examples

Example 1	*Example 2*	*Example 3*
DIM myArray**(5)** AS INTEGER FOR **loops** 1 TO **10** myArray(**loops**) = 0 NEXT index	DIM score AS **INTEGER** **score** = “**Henry**”	DIM result AS INTEGER DIM value AS INTEGER **value = 0** result = 0 result = **50 / value**
Error: Trying to access an array position bigger than the size of the array. **Explanation:** The value of the **loops** variable will increase until it gets to 10. When it reaches 6, this will exceed the size of the array which only has 5 positions.	**Error:** Trying to assign string data to an integer variable. **Explanation:** The score variable has been declared as an Integer data type. When the program tries to store a string value in score, it cannot be done.	**Error:** Trying to divide by 0 which is impossible. **Explanation:** The value variable has been initialised to 0. When the program tries to divide 50 by the contents of value (0), it cannot be done.

Logic Error

A logic error occurs when the program runs normally, does not crash, but produces an incorrect output (the wrong answer).

A logic error will **not** cause the program to crash. These type of errors can be difficult to fix because **no error messages** are produced.

Examples

total = score1 ***** score2
 	
Msgbox(“score1 + score2 is ” & total)
	IF answer1 = “A” **AND** answer1= “B” THEN
	        result = “pass”
	ELSE
	        result = “fail”
	END IF

The expression multiplies the scores instead of adding them, giving the wrong output. The first line should be: total = score1 **+** score2	The complex condition (line 1) is checking for answer1 to contain A and B at the same time. It is impossible for a variable to contain both A and B so the result will always be FAIL. The first line should use an OR instead of AND: IF answer1 = “A” **OR** answer1= “B” THEN

TEST DATA

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

The following examples are slightly different as they deal with real world inputs. Normal data is any data that is possible (but not those on the boundary where a change takes place).

A sensor detects the speed of a car and sends an alert if it is travelling over 70mph

Normal data in this case would be:
0 to 69 and **72 to 150 approx**

A sensor detects the temperature in a school and turns the heating on if it drops to 18^{o} or below.

Normal data in this case would be:
approx -50 to 17 and **20 to 60 approx**

It could be possible for a car to travel at 0mph or 150mph so these are normal values

It could be possible for the temperature to be -50^{o} or 60^{o} so these are normal values

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 and **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 and 100

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

Extreme data in this case would be:
0 and 50

The following examples are slightly different as they deal with real world inputs. Extreme data is any data that is on the boundary between one event and another.

A sensor detects the speed of a car and sends an alert if it is travelling over 70mph

Extreme data in this case would be:
70 and **71**

70 is the last value that does not cause the alert to be sent.

71 is the first value that causes the alert to be sent

A sensor detects the temperature in a school and only turns the heating on if it is 18° or below.

Extreme data in this case would be:
18 and **19**

18 is the last temperature that causes the heating to be on.

19 is the first temperature that causes the heating to be off.

Exceptional Test Data

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**

Exceptional 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**

The following examples are slightly different as they deal with real world inputs. Extreme data is any data that is on the boundary between one event and another.

A sensor detects the speed of a car and sends an alert if it is travelling over 70mph

Extreme data in this case would be:
-100, 1000000, "Fred"

A sensor detects the temperature in a school and only turns the heating on if it is 18^{o} or below.

Extreme data in this case would be:
-500, 1000, "Jane"

These values are impossible speeds for a car to be travelling at.

Entering a text value when a number is expected is also exceptional.

These values are impossible temperatures for a school to be at.

Entering a text value when a number is expected is also exceptional.

TEST TABLE

A test table is used to record tests carried out, expected results and actual results. A test table should use all three types of test data.

Expected results are worked out **manually** without using the program. The same inputs are then entered into the program to give the **actual results**. If both match then the test has passed.

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 Result
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

Test 5 has failed because the program has accepted invalid score inputs (all above 50) when it should have asked the user to re-enter.

READING REVIEW 8

Having read pages 59 – 66, answer the questions below in preparation.

1. Using examples, explain the following terms:

Normal Test Data

Extreme Test Data

Exceptional Test Data

Syntax Error

Execution Error

Logic Error

EVALUATION

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

An evaluation report would discuss:

- Fitness for Purpose
- Robustness
- Efficient use of coding constructs
- Readability

FITNESS FOR PURPOSE

A program is fit for purpose if it:

- Carries out all the **functional requirements** from the analysis stage
    - Can the program do everything is was expected to do?
    - Are all of its expected processes present and working?

- Passes all the **tests** carried out at the testing stage.
    - Do all tests produce the correct expected outputs?

ROBUSTNESS

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

> ***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 (e.g. twelve). If the program was to crash then it would not be a robust program.

Instead of crashing, a robust program will use **input validation** to handle the error and provide a helpful message to the user.

EFFICIENT USE OF CODING CONSTUCTS

An efficient program is one which carries out **only as many instructions as necessary** to complete its purpose.

There are a number of ways to make code more efficient and reduce how many lines of code are carried out.

Construct: Repetition

The use of repetition can greatly reduce the number of lines of code that have to be typed.

This program finds the average of 10 numbers.

```
DECLARE total INITIALLY 0
RECEIVE number FROM KEYBOARD
SET total TO total + number
RECEIVE number FROM KEYBOARD
SET total TO total + number
RECEIVE number FROM KEYBOARD
SET total TO total + number
RECEIVE number FROM KEYBOARD
SET total TO total + number
RECEIVE number FROM KEYBOARD
SET total TO total + number
SET average TO total / 5
```

This program does exactly the same but uses more efficient constructs.

```
DECLARE total INITIALLY 0
REPEAT 10 TIMES
      RECEIVE number FROM KEYBOARD
      SET total TO total + number
END REPEAT
SET average = total / 10
```

This can also be done using a 1D array.

Without an array, multiple variables are needed to store all the values.

```
DECLARE total INITIALLY 0
RECEIVE number1 FROM KEYBOARD
RECEIVE number2 FROM KEYBOARD
RECEIVE number3 FROM KEYBOARD
RECEIVE number4 FROM KEYBOARD
RECEIVE number5 FROM KEYBOARD

SET total TO number1 + number2 +
number3 + number4 + number5

SET average TO total / 5
```

An array allows all the numbers to be stored in a single data structure.

```
DECLARE number INITIALLY []
FOR counter FROM 1 TO 5 DO
      RECEIVE number[counter)]
      SET total TO total + number[counter]
END FOR
SET average TO total/5
```

Construct: Selection

Choosing from a number of possible alternatives when using selection can make code more efficient, however, it is not always obvious which is more efficient.

These two programs decide the grade that a candidate is given, depending on the mark they received in the exam.

<table>
<tr>
<td>This uses four <b>separate IF</b> constructs, one after another, with the use of complex conditional statements.
<pre>
IF mark < 50 THEN
      SET grade TO D
END IF

IF mark>=50 AND mark<=59 THEN
      SET grade TO C
END IF

IF mark>=60 AND mark<=69 THEN
      SET grade TO B
END IF

IF mark>=70 THEN
      SET grade TO A
END IF
</pre></td>
<td>This uses <b>nested IF</b> constructs with simple conditional statements.
<pre>
IF mark>=70 THEN
      SET grade=A
ELSEIF mark>=60 THEN
      SET grade=B
ELSEIF mark>=50 THEN
      SET grade=C
ELSE
      SET grade=D
END IF
</pre></td>
</tr>
<tr>
<td>This program always carries out four comparisons, regardless of the values stored in mark.</td>
<td>This program carries out either one, two or three comparisons, depending on the values stored in the mark.</td>
</tr>
</table>

Construct: Logical Operators

Logical operators can be useful when creating complex conditions, rather than using multiple simple conditions.

<table>
<tr>
<td>This program uses two <b>simple</b> conditional statements.
<pre>
IF X > 4 THEN
      IF Y < 6 THEN
            SET quadrant TO 2
      END IF
END IF
</pre></td>
<td>This program uses one <b>complex</b> conditional statement.
<pre>
IF X &gt; 4 <b>AND</b> Y &lt; 6 THEN
      SET quadrant TO 2
END IF
</pre></td>
</tr>
</table>

READABILITY

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

Some methods of making a program readable are:

- Use meaningful identifiers (variable names)
- Use internal commentary
- Use indentation
- Use plenty of white space

Meaningful Identifiers

Giving variables sensible names makes it easier to identify what they are being used to store.

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. DECLARE **x** INITIALLY 0 DECLARE **y** INITIALLY 0 DECLARE **z** INITIALLY 0 SET **x** TO **y** * **z**	It is much better to use sensible variable names like **total**, **PupilCost** and **pupils**. DECLARE **total** INITIALLY 0 DECLARE **pupilCost** INITIALLY 0 DECLARE **pupils** INITIALLY 0 SET **total** TO **pupilCost** * **pupils**

Internal Commentary

Internal commentary is used to provide 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
'01/04/2018

'declare variables
Dim total as Single
Dim PupilCost as Single
Dim pupils as Integer

'calculate total cost
total = PupilCost * Pupils
```

Indentation

Indentation 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 selection constructs.

Without indentation, it is difficult to see the content of IF statements...

```
IF mark < 50 THEN
SET grade TO D
END IF
IF mark>=50 AND mark<=59 THEN
SET grade TO C
END IF
```

…or loops.

```
REPEAT 10 TIMES
RECEIVE number FROM KEYBOARD
SET total TO total + number
END REPEAT
```

Using indentation, the start and end IF statements...

```
IF mark < 50 THEN
      SET grade TO D
END IF
IF mark>=50 AND mark<=59 THEN
      SET grade TO C
END IF
```

…and loops is much clearer.

```
REPEAT 10 TIMES
      RECEIVE number FROM KEYBOARD
      SET total TO total + number
END REPEAT
```

White Space

White space is used to make code more readable by leaving blank lines between main steps of the program.

This makes it easier to identify the different sections of the program.

Without white space, it is difficult to pick out the different stages of the program.

```
DECLARE pupilName INITIALLY “”
DECLARE pupilAge INITIALLY 0
DECLARE number INITIALLY []
RECEIVE pupilName FROM KEYBOARD
RECEIVE pupilAge FROM KEYBOARD
FOR counter FROM 1 TO 5 DO
      RECEIVE score[counter)]
      SET total TO total +
      score[counter]
END FOR
SET average TO total/5
```

White space helps to separate the different stage.

```
DECLARE pupilName INITIALLY “”
DECLARE pupilAge INITIALLY 0
DECLARE number INITIALLY []

RECEIVE pupilName FROM KEYBOARD
RECEIVE pupilAge FROM KEYBOARD


FOR counter FROM 1 TO 5 DO
      RECEIVE score[counter)]
      SET total TO total +
      score[counter]
END FOR
SET average TO total/5
```

READING REVIEW 9

Having read pages 60 – 72, answer the questions below in preparation.

1. Explain the term fit for purpose.

2. Explain the term robustness.

3. Explain the term readability.

4. Why is it important to have readable code?

5. State four ways in which code can be made more readable.