Computing Science

Software Design & Development

Theory Notes

Contents

Development Methodologies

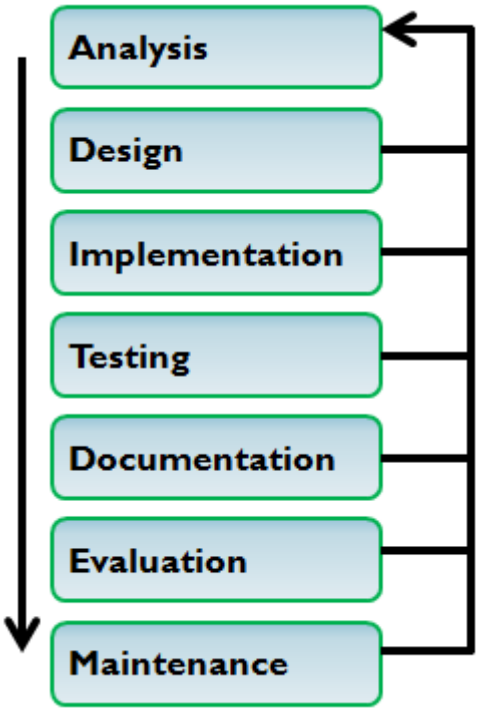
Development Methodologies

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.



Software development is an **iterative process**.

At any point, it may be necessary to **revisit earlier stage**s.

This could be in order to make improvements due to new information or the presence of errors.

Top-Down Design / Stepwise Refinement

Top-down design involves identifying an overall problem and breaking it down into smaller sub-problems (main steps).

The process of stepwise refinement is then used to break the sub-problems down until each one is small enough that they are manageable.

Top-down design emphasises planning and a complete understanding of the system.

No coding should take place until a sufficient level of detail has been reached in the design.

Each sub-problem is coded as a module however this delays testing of the functional units until significant design is complete.

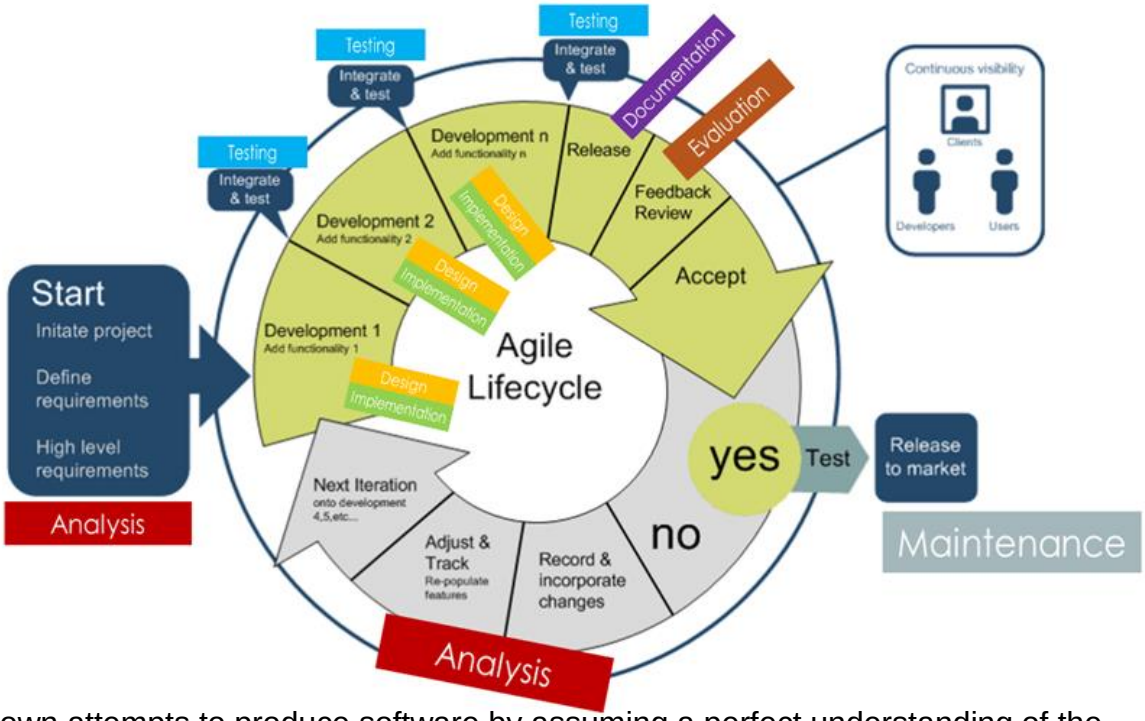
Agile Methodologies

Agile development emphasises real time, **face-to-face communication** involving all the people necessary to finish the software.

Very little written documentation is produced.

Software is developed in short iterations, each one like a miniature software project of its own.

The purpose of a single iteration is not to produce the final completed solution, but to add additional functionality that produces working software. After each iteration the project priorities are then re-evaluated.



Top-down attempts to produce software by assuming a perfect understanding of the client's requirements from the start. In reality however, it rarely delivers what the client wants as the client often doesn't know exactly what they want until they see it.

Agile methodologies embrace **iterations** (Sprints) where small teams work to develop working software that builds on the previous iteration.

During each iteration, working software is produced following which the requirements for the next iteration can be evaluated.

Sprints

- A sprint is a planned delivery schedule for an aspect of the system. Within a sprint the principles of analysis, design, implementation and testing are used.
- Prototyping is also likely, particularly during the early phase of a sprint.
- Sprints are carried out for each area of development, so rather than having a rigid set of steps to follow for the development of the entire system, several steps are repeated in one sprint and then carried out again in the next sprint.

Agile Disadvantages

- The main drawback of agile methods is that following a sequence of sprints and engaging in near daily communication is very time consuming.
- The emphasis on team work and communication in a face to face manner means that long term, large scales projects are often unrealistic.
- Agile methods tend to suit small scale development better than large scale development.

Iterative vs Agile

	Iterative (Waterfall Model)	**Agile**
Client Interaction	The client is heavily involved in the initial analysis stage and at the end of development, when evaluating if the software meets their needs and matches the agreed specification.	The client is involved throughout the process, giving constant feedback on prototypes of the software during development. This feedback is acted upon, quickly ensuring the software evolves throughout the project. Changing goals during the development can be positive in terms of final client satisfaction with the product.
Teamwork	Teams of analysts, programmers, testers and documenters work independently on each phase of development. Teams mainly work in isolation with some communication required between each phase.	Teams of developers communicate and collaborate, rather than teams of experts operating in isolation. During a project, fast, face-to-face communication between individuals with different skills is an important factor in progressing the project quickly.
Documentation	A detailed project specification is created at the beginning of a project. Significant time is spent during the project on design, program commentary and test plans.	While modelling solutions remains important, creating large documents that are never updated or referred to again upon completion of the project are not. Agile focuses on reducing documentation. It spends time on small cycles of coding, testing and adapting to change. Any documentation produced (for example internal commentary in code) should focus purely on progressing the project.

Measurement of progress	Follows a strict plan, with progress measured against timescales set at the beginning of the project.	Breaks a project down into a series of short development goals (often called “sprints”). This involves cross-functional teams working on: planning, analysis, design, coding, unit testing, and acceptance testing. Progress is measured by the time it takes to produce prototypes or working components of the software. Agile focuses on delivering software as quickly as possible.
Adaptive vs predictive	A predictive methodology, focusing on analysing and planning the future in detail and catering for known risks. Predictive methods rely on effective early phase analysis and if this goes very wrong, the project may have difficulty changing direction. Predictive teams often institute a change control board to ensure they consider only the most valuable changes.	An adaptive methodology, focusing on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team has difficulty describing exactly what they will do next week but could report on which features they plan for next month. The further away a date is, the vaguer an adaptive method is about what will happen on that date.
Testing	Testing is carried out when the implementation phase of the project is complete.	There is no recognised testing phase, as testing is carried out in conjunction with programming.

Analysis

Analysis Stage

This is the start of the software development process and defines the extent of the software task. This is called the software specification. It is often the basis of a legal contract between the client (customer) and the software company writing the software.

Your analysis should include the following:

- **Purpose**: a general description of the purpose of the software.
- **Scope**: a list of the deliverables that the project will hand over to the client and/or end-user, eg design, completed program, test plan, test results and evaluation report. It can also include any time limits for the project.
- **Boundaries**: the limits that help to define what is in the project and what is not. It can also clarify any assumptions made by the software developers regarding the client's requirements.
- **Functional requirements**: the features and functions that must be delivered by the system in terms of inputs, processes and outputs.

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.

Example:

Purpose

The purpose of this program is to take 20 pupil names, their prelim marks and their assignment marks from a file. Calculate the percentage, and then find and display the name and percentage of the pupil with the highest percentage.

Scope

This development involves creating a modular program. The deliverables include:

- detailed design of the program structure
- test plan with completed test data table
- working program
- results of testing
- evaluation report

This development work must be completed within 4 hours.

Boundaries

The program will read the pupil data (name, prelim mark and assignment mark) for 20 pupils from a sequential file. The data is accurate, so there is no need to implement input validation.

The pupil with the top mark will be the pupil who has the highest percentage. The only output needed is the name and percentage of the pupil with the highest percentage.

Functional Requirements

These are defined in terms of the inputs, processes and outputs detailed below. All inputs are imported from a sequential file and all outputs displayed on the screen. The program is activated by double clicking on the file icon and then selecting “Run” from the menu. Each process should be a separate procedure or function that is called from the main program.

Inputs: Pupil name
Prelim mark
Assignment mark

Processes: Calculate the percentage for each pupil
Find the name and percentage of the pupil with the highest percentage

Outputs: Name of the pupil with the highest percentage
The highest percentage

Design

Design Techniques

Pseudocode

When using pseudocode to design efficient solutions to a problem, you must include the following:

- **Top level design** — the major steps of the design. In the example below, numbered from 1 to 4.

- **Data flow** — shows the information that must flow In or Out from the sub-programs. In the example below, written to the right of the top level design.

- **Refinements** — break down the design from the top level when required. In the example below, numbered as a sub-number of the top level.

Example:

The following design is for a program that will read the name, prelim mark and coursework mark for a class of 20 pupils from a file. It will calculate a percentage from each of their prelim marks and coursework marks added together. It will then display the name of the pupil with the highest percentage and their percentage.

1 Get results (**OUT**: pupil name(), prelim mark(), course mark())
2 Calculate percentages **(IN**: prelim mark(), course mark() **OUT**: percentage())
3 Find position of pupil with top mark (**IN**: percentage() **OUT**: top position)
4 Display pupil with top mark (**IN**: pupil name(), top position)

1.1 Open marks file
1.2 Start fixed loop for each pupil
1.3 Get pupil name()
1.4 Get prelim mark()
1.5 Get course mark()
1.6 End fixed loop
1.7 Close marks file

2.1 Start fixed loop for each pupil
2.2 percentage() equals (prelim mark() + course mark()) divided by 1.5
2.3 End fixed loop

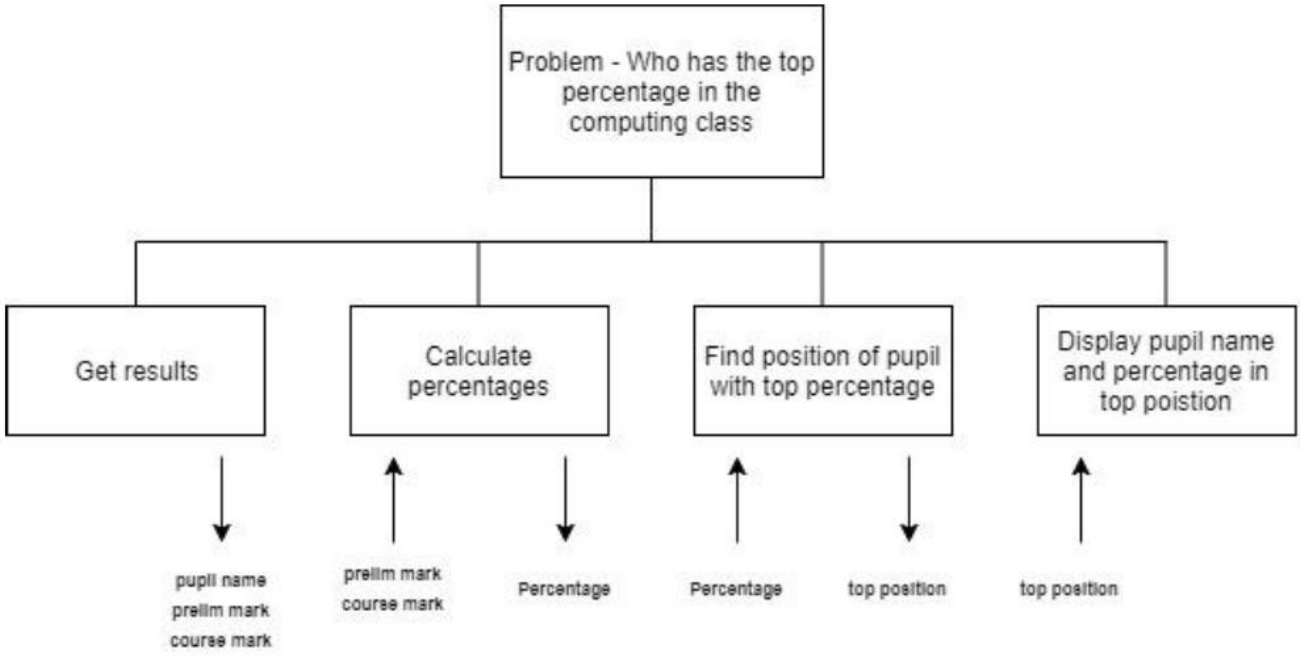
3.1 top position equals first position
3.2 Start fixed loop from second pupil
3.3 If percentage() is greater than current top percentage Then
3.4 set position as new top position
3.5 End If
3.6 End fixed loop

4.1 Display “Top pupil is”, pupil name(top position), “with”, percentage(top position), “percent”

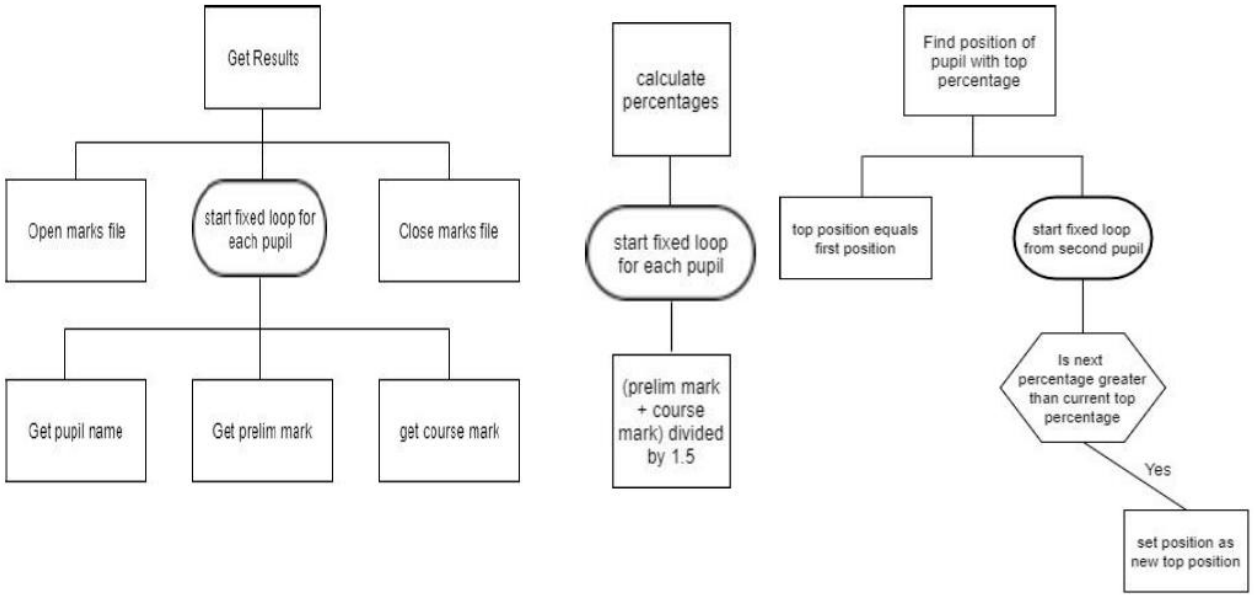
Structure diagrams

The following structure diagram solves the same problem as the pseudocode:

- **Top level design** — the major steps of the design.
- **Data flow** — shows the information that must flow In or Out from the sub-programs. In the example below, written underneath the top level design with an arrow showing whether they are in or out.



- **Refinements** — break down the design from the top level into smaller steps. They can be shown separately from the top level design or below the top level design.



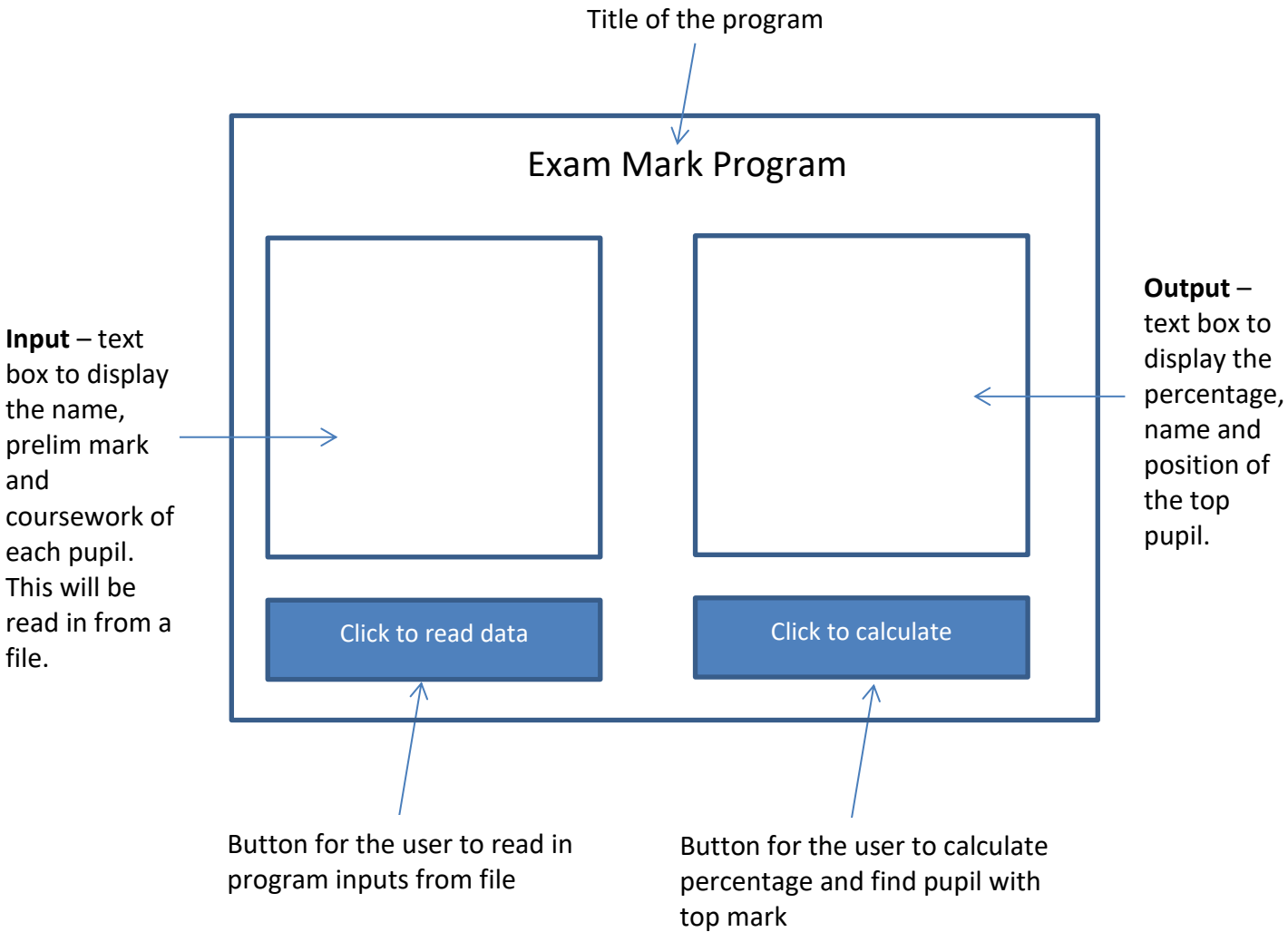
Wireframe (User Interface Design)

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.

The wireframe diagram should clearly show the program input and output.



Implementation

Sub-Programs

Sub-programs are named blocks of code which can be run from within another part of the program.

When a sub-program is used like this we say it is “called”.

Sub-programs can be called from any part of the program and can be used over again.

A sub-program may be called several times during the execution of a single program.

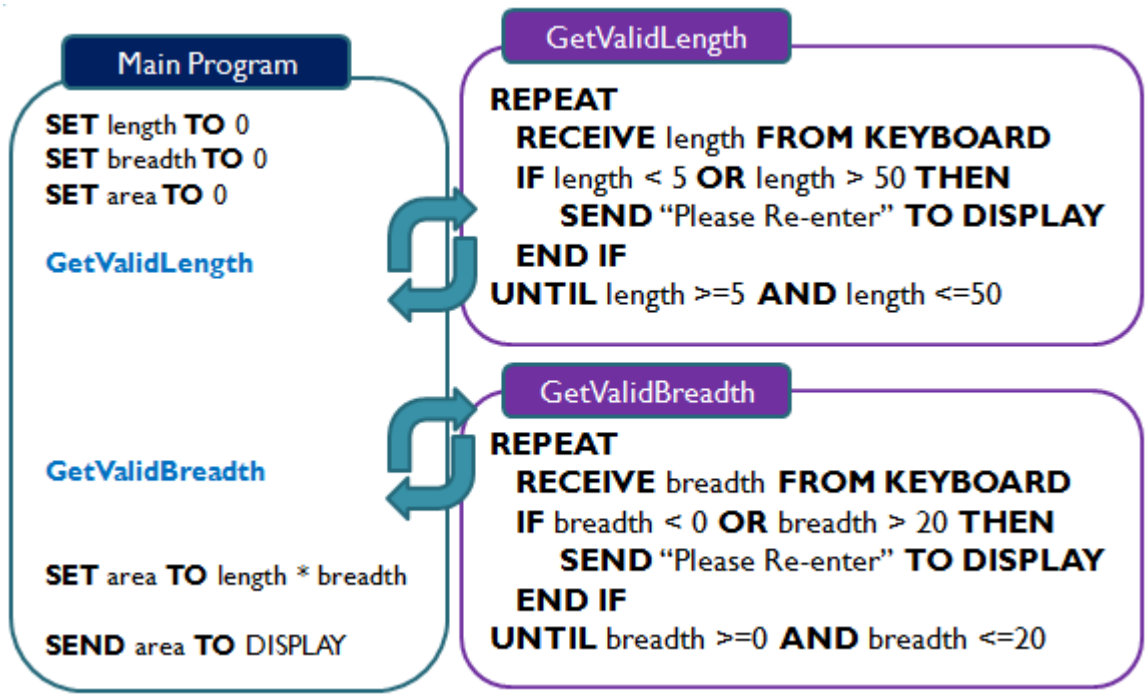
Example

This program works out the area of a room in a building.

Main Program

```
SET length TO 0
SET breadth TO 0
SET area TO 0

REPEAT
  RECEIVE length FROM KEYBOARD
  IF length < 5 OR length > 50 THEN
     SEND “Please Re-enter” TO DISPLAY
  END IF
UNTIL length >=5 AND length <=50

REPEAT
  RECEIVE breadth FROM KEYBOARD
  IF breadth < 0 OR breadth > 20 THEN
     SEND “Please Re-enter” TO DISPLAY
  END IF
UNTIL breadth >=0 AND breadth <=20

SET area TO length * breadth
SEND area TO DISPLAY
```

The Input Validation lines of code can be put into **sub-programs** and **called** by the main program.



Why Create Sub-Programs?

Creating sub-programs makes the code more **modular** and readable.

Modular code allows sections of code to be self-contained.

Different sub-programs can be developed by different programmers without variable name clashes

Sub-programs can be re-used without any extra coding which saves time.

Easier to identify errors.

Types of Sub-Program

There are two types of sub-program that can be used in procedural languages.

- **Procedures**
- **Functions**

Procedures and functions are self-contained sections of code that execute a sequence of commands.

They are both given meaningful identifiers (names) which are used to call them.

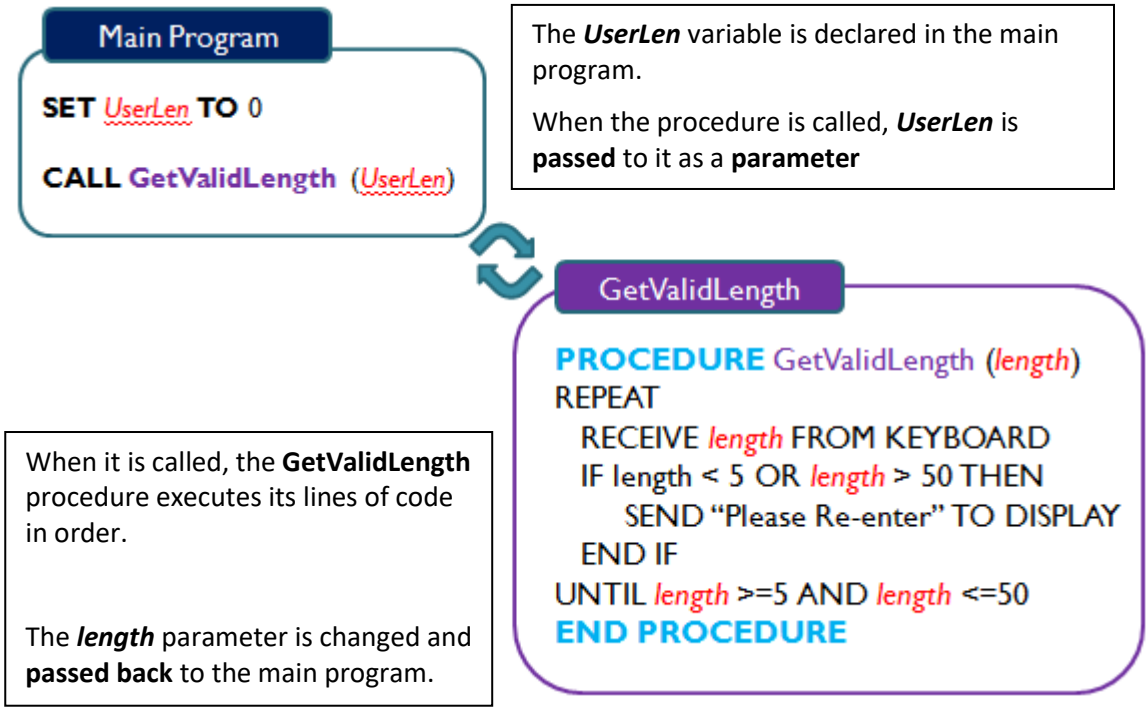
Procedures

When procedures are called, variables (parameters) to be passed **in or out** of the procedure are **stated in brackets**.

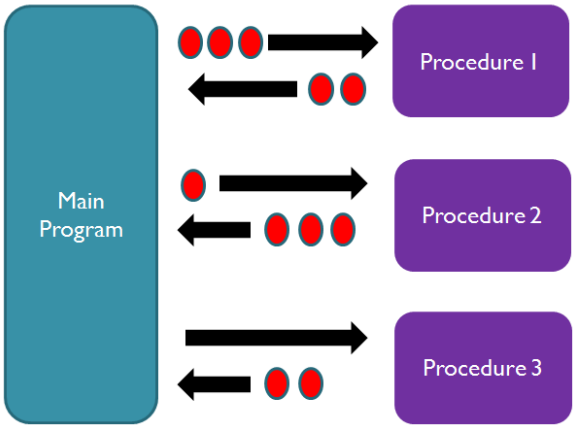
Procedures can pass **any number of parameters** in or out (or sometimes none).

Example

Consider creating the GetValidLength sub-program as a **procedure.**



Any number of parameters (variables) can be passed **in or out** of procedures.



Functions

When functions are called, variables (parameters) to be passed **in only** are **stated in brackets**.

Functions can **return** only a **single value.**

The returned value from a function is **assigned to a variable** to be used in subsequent operations in the program.

Example

Consider creating the GetValidLength sub-program as a **function**

MainProgram

```
SET UserLen TO 0
UserLen = GetValidLength( )
```

The ***UserLen*** variable is declared in the main program.

When the function is called, ***UserLen*** is used to store the **returned integer**

When it is called, the **GetValidLength** function executes its lines of code in order.

The ***length*** variable is **returned** to the main program and stored in ***UserLen***

GetValidLength

```
FUNCTION GetValidLength ( ) RETURNS INTEGER
REPEAT
  RECEIVE length FROM KEYBOARD
  IF length < 5 OR length > 50 THEN
     SEND "Please Re-enter" TO DISPLAY
  END IF
UNTIL length >=5 AND length <=50
RETURN length
END FUNCTION
```

Re-Using Sub-Programs

The most efficient use of sub-programs is when they can be re-used .

When coding procedures and functions, consideration should be given to making them able to solve **any related problem** rather than **one specific problem**.

e.g. a calculator that can only solve the calculation 2+2 would be very limited.

Example

The procedures below are almost identical except for the range of values they validate.

GetValidLength

```
PROCEDURE GetValidLength (length)
REPEAT
  RECEIVE length FROM KEYBOARD
  IF length < 5 OR length > 50 THEN
     SEND "Please Re-enter" TO DISPLAY
  END IF
UNTIL length >=5 AND length <=50
END PROCEDURE
```

GetValidBreadth

```
PROCEDURE GetValidBreadth (breadth)
REPEAT
  RECEIVE breadth FROM KEYBOARD
  IF breadth < 0 OR breadth > 20 THEN
     SEND "Please Re-enter" TO DISPLAY
  END IF
UNTIL breadth >=0 AND breadth <=20
END PROCEDURE
```

They could instead be made more generic by allowing the range of values to be changed each time it is called.

GetValidValue

```
PROCEDURE GetValidValue (low, high, userVal)
REPEAT
  RECEIVE userVal FROM KEYBOARD
  IF userVal < low OR userVal > high THEN
     SEND "Please Re-enter" TO DISPLAY
  END IF
UNTIL userVal >=low AND userVal <=high
END PROCEDURE
```

The GetValidValue procedure can now be called to obtain a value within **any range** specified.

Main Program

```
SET UserLen TO 0
SET UserBre TO 0

CALL GetValidValue (5, 50, UserLen)

CALL GetValidValue (0, 20, UserBre)
```

GetValidValue

```
PROCEDURE GetValidValue (low, high, userVal)
REPEAT
   RECEIVE userVal FROM KEYBOARD
   IF userVal < low OR userVal > high THEN
      SEND "Please Re-enter" TO DISPLAY
   END IF
UNTIL userVal >=low AND userVal <=high
END PROCEDURE
```

The implementation of a reusable **function** would look like this.

Main Program

```
SET UserLen TO 0
SET UserBre TO 0

UserLen = GetValidValue (5, 50)

UserBre = GetValidValue (0, 20)
```

GetValidValue

```
FUNCTION GetValidLength (low, high) RETURNS INTEGER
REPEAT
   RECEIVE userVal FROM KEYBOARD
   IF userVal < low OR userVal > high THEN
      SEND "Please Re-enter" TO DISPLAY
   END IF
UNTIL userVal >= low AND userVal <= high
RETURN userVal
END FUNCTION
```

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.

Function	Purpose
Int	Convert floating point numbers (decimal) to integers
ASC	Converts a character into its corresponding ASCII code value (e.g. "A" into 65)
Char	Convert a number into the corresponding ASCII character (e.g. 65 into "A")
Mod	Divides two numbers and only returns the remainder
Substring	Function used to extract a substring from a string

See practical notes on how to implement the above predefined functions.

Parameter Passing

What is parameter passing?

Parameters are

- the **variables** or **arrays** that are passed in or out of procedures.
- the **variables** or **arrays** that are passed into functions

Parameter passing allows variables to be **used and updated** by sub-programs.

The program below uses three procedures.

Main Program

```
SET UserLen TO 0
SET UserBre TO 0
SET TotalArea TO 0

CALL GetDimensions (UserLen, UserBre)
CALL CalculateArea (UserLen, UserBre, Area)
CALL DisplayArea (TotalArea)
```

The variables in brackets are **parameters** required by the procedures stored in ***UserLen***

These parameters are sent to the procedures to be used.

Each parameter is known as an **argument**.

Procedures must be **declared** before they are **called**.

Main Program

```
...
CALL GetDimensions (UserLen, UserBre)
...
```

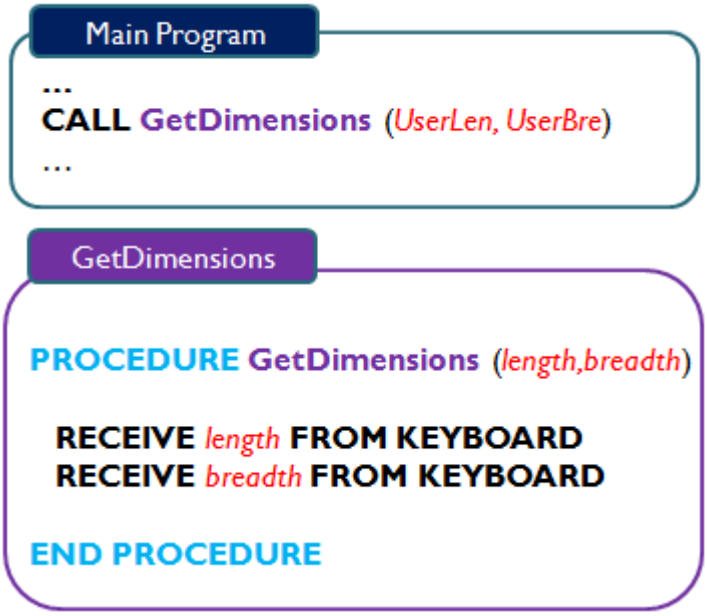
This line **CALLS** the procedure

GetDimensions

```
PROCEDURE GetDimensions (length,breadth)

  RECEIVE length FROM KEYBOARD
  RECEIVE breadth FROM KEYBOARD

END PROCEDURE
```

This section **DECLARES** the procedure

The **order** in which parameters are listed is important.

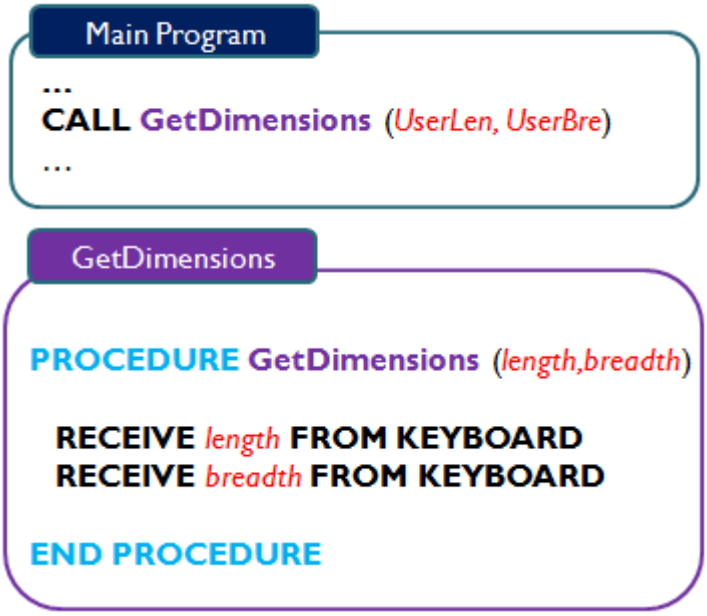


Notice the **order** of the parameters here…

…must be the same as the order here – but the **names can be different**

Actual and Formal Parameters?

Parameters can be **actual** or **formal**.



These are known as **actual parameters**

These are known as **formal parameters**

Actual parameters contain the value which is to be passed to the sub-program's formal parameter.

Formal parameters are used by the sub-program and contain a copy of or link to the values passed from the actual parameters.

Main Program

```
CALL GetDimensions (UserLen, UserBre)
CALL CalculateArea (UserLen, UserBre, Area)
CALL DisplayArea (TotalArea)
```

GetDimensions

```
PROCEDURE GetDimensions (length,breadth)

  RECEIVE length FROM KEYBOARD
  RECEIVE breadth FROM KEYBOARD

END PROCEDURE
```

CalculateArea

```
PROCEDURE CalculateArea (length, breadth,Area)

  SET Area TO length * breadth

END PROCEDURE
```

DisplayArea

```
PROCEDURE DisplayArea (Area)

  SEND Area TO DISPLAY

END PROCEDURE
```

Scope of Variables

The scope of a variable is the area of code in which the variable is usable

i.e. how much of the program has access to it.

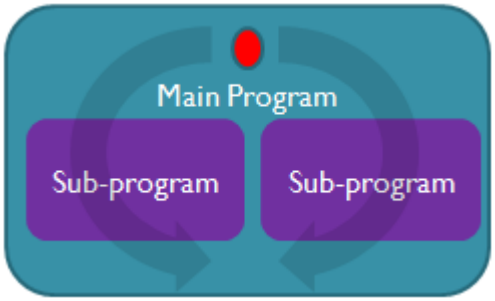
The scope of a variables can be either:

- **Global**
- **Local**

Global Variables

A global variable exists and can be accessed and changed from **any part of the program.**

Global variables do not have to be passed into procedures as parameters because the procedure can access it without doing so.



Global variables **reduce modularity** of a program and should be avoided wherever possible.

The use of global variables reduces modularity because:

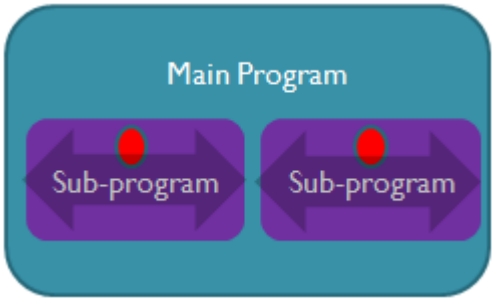
- Different programmers could use **conflicting variable names** which would cause errors.
- Any procedure could **accidentally alter** a global variable as it doesn't have to be passed in to be used.

Local Variables

Local variables exist only **within a procedure** or **function**. They are **declared within** a sub-program

They are **not passed in or out** and can only be used within the sub-program they were declared in.

Local variables **cannot be accessed** from out with their own sub-program which limits their scope.



It is always preferable to limit the scope of a variable to an individual sub-program wherever possible.

Limiting the scope of a variable is done by:

- Using **local variables** which can only be accessed with their own sub-program.
- Using **parameter passing** to only pass to a sub-program the variables it requires.

Data Types & Structures

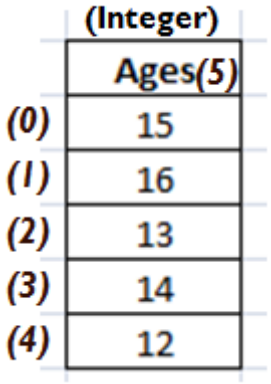
Data Types

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

Data Type	Contents	Example
CHARACTER	Single Letter	"A","B","C"
INTEGER	Whole Number	-1,-12, 15, 18, 100
SINGLE	Real Number	2.45, 3.9, 12.994
BOOLEAN	True/False	TRUE, FALSE

Data Structures: 1 Dimensional Array

A 1D array is an ordered sequence of simple data types, all of the same type.

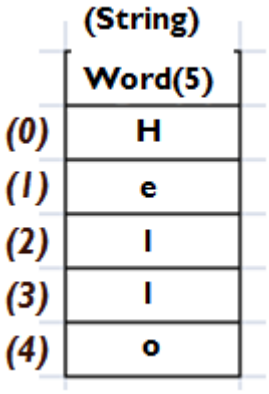


Advantages over separate individual variables:

- Only one line of code required to create multiple values
- Can be traversed using a loop structure
- Parameter passing is easier
- Indexing allows each individual element to be referenced

Data Structures: String

A string is a special sort of array that contains characters. A string is actually a just a list of single characters.

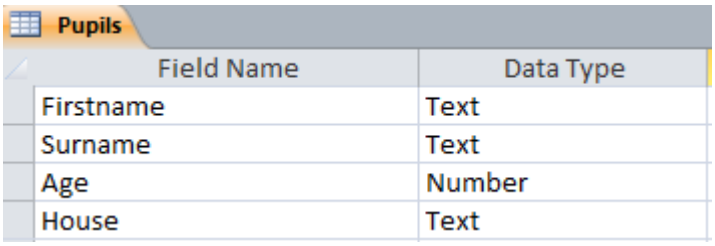


Strings can be joined using concatenation or extracted using substrings.

Data Structures: Records / Array of Records

Records are **customised data types** created by the programmer. They can contain **several variables** which can be of **different data types**.

When you create a record structure, you are essentially creating a database structure.



Pupils

Field Name	Data Type
Firstname	Text
Surname	Text
Age	Number
House	Text

A **record structure** is created by giving the structure a name and defining the 'fields' required.

```
RECORD recordname IS
{datatype fieldname1, datatype fieldname2, datatype fieldname3…}
```

An **array of records** is then declared which specifies the size of the array and the record structure to use (as the data type):

```
DECLARE arrayname(indexes) AS recordname
```

Notice, instead of declaring the array using a data type such as integer or string, we have used the name of the record structure as the data type.

Example:

Create a record structure to store the information below for 10 pupils:

First Name	Surname	Age	House
Harry	Jones	14	Bute
Jenna	White	12	Kintyre
Laura	Cairns	15	Arran

a) Defining the **record structure**:

```
RECORD  Userdetails  IS
{STRING Firstname, STRING Surname, INTEGER Age, STRING House}
```

b) Declare an **array of records**.

```
DECLARE UserRecord(10) As Userdetails
```

Record values can now be initialised or updated as a complete record

```
SET UserRecord[1] TO {“Harry”, “Jones”, 37}
```

Or by referring to individual values

```
SET UserRecord[1].Firstname TO “Harry”

SET UserRecord[1].Surname TO “Jones”

SET UserRecord[1].Age TO 37
```

Standard Algorithms

Standard Algorithms (Using Arrays)

Standard algorithms are sequences of code which are used regularly in different programs to solve a particular problem.

The main algorithms you need to know at Higher are:

- Input Validation *(N5 revision)*
- Find Minimum
- Find Maximum
- Count Occurrences
- Linear Search

Find Minimum

Find Minimum is used to identify the smallest value in an array.

If used on the Ages array (*right*), Find Minimum would return the value, 7

Ages	
0	15
1	12
2	7
3	16
4	12

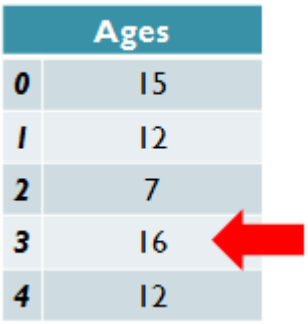
```
1. SET min TO ages[0]

2. FOR counter FROM 1 TO 4 DO
3.         IF ages[counter] < min
4.                 SET min TO ages[counter]
5.         END IF
6. END FOR

7. SEND "The lowest age is "& min TO DISPLAY
```

1.Min is initialised to match first item in array
2.Repeat for each item in array starting at item 2
3.Check if current array item is lower than Min
4.If true, set Min to match current array item

Find Maximum

Find Maximum is used to identify the largest value in an array.

If used on the Ages array above, Find Maximum would return the value, 16



Ages	
0	15
1	12
2	7
3	16
4	12

```
1. SET max TO ages[0]

2. FOR counter FROM 1 TO 4 DO
3.         IF ages[counter] > max
4.                 SET max TO ages[counter]
5.         END IF
6. END FOR

7. SEND "The highest age is "& max TO DISPLAY
```

1.Max is initialised to match first item in array
2.Repeat for each item in array starting at item 2
3.Check if current array item is higher than Max
4.If true, set Max to match current array item

Count Occurrences

Count Occurrences is used to identify how many times a particular value appears in an array

If used on the Names array above for the name "Betty", Count Occurrences would return the value, 2

	Names
0	Fred
1	Betty
2	Wilma
3	Betty
4	Barney

Betty
2

```
1. RECEIVE target FROM KEYBOARD
2. SET numFound TO 0

3. FOR counter FROM 0 TO 4
4.       IF names[counter] = target
5.                SET numFound TO numFound + 1
6.       END IF
7. END FOR

8. SEND "The number found is "& numFound TO DISPLAY
```

1. Ask user to enter target value to count
2. Initialise numFound to 0
3. Repeat for each item in array
4. Check if current name matches target
5. If true, **increment** numFound by 1

Linear Search

Linear Search is used to identify whether or not an item is in a list, and which position it occupies.

If used on the Names array above for the name, “Betty”, Linear Search would return the position, 3

	Names
0	Fred
1	Betty
2	Wilma
3	Betty
4	Barney

Betty
FOUND
at position 3

```
1. RECEIVE target FROM KEYBOARD
2. SET found TO FALSE
3. SET positionTO 0

4.   FOR counter FROM 0 TO 4
5.      IF names[counter] = target
6.          SET found TO TRUE
7.          SET position TO counter
8.      END IF
9.   END FOR

10. IF found = TRUE
11.    SEND “Found at position ”& position TO DISPLAY
12. ELSE
13.    SEND “Not found”
14. END IF
```

1. Ask user to enter target value to find
2-3. Initialise found **“flag”** to False and position to 0
4. Repeat for each item in array
5. Check if current name matches target
6-7. If true, set found **“flag”** to true and position to current loop value

Standard Algorithms (Using Record Structures)

Find Minimum

Find Minimum is used to identify the smallest value in an array of record structure.

If used on the UserDetails record structure below, Find Minimum would return the value, 12.

RECORD *Userdetails* **IS**
{**STRING** Firstname, **STRING** Surname, **INTEGER** Age, **STRING** House}

First Name	Surname	Age	House
Harry	Jones	14	Bute
Jenna	White	12	Kintyre
Laura	Cairns	15	Arran
Sam	Kay	16	Arran
Harry	Smith	14	Lomond

```
1. SET min TO UserDetails[0].ages

2. FOR counter FROM 1 TO 4 DO
3.         IF UserDetails [counter].ages < min
4.                 SET min TO UserDetails [counter].ages
5.         END IF
6. END FOR

7. SEND "The lowest age is "& min TO DISPLAY
```

Find Maximum

Find Maximum is used to identify the largest value in an array of record structure.

If used on the UserDetails record structure below, Find Maximum would return the value, 16.

RECORD *Userdetails* **IS**
{**STRING** Firstname, **STRING** Surname, **INTEGER** Age, **STRING** House}

First Name	Surname	Age	House
Harry	Jones	14	Bute
Jenna	White	12	Kintyre
Laura	Cairns	15	Arran
Sam	Kay	16	Arran
Harry	Smith	14	Lomond

```
1. SET max TO UserDetails [0].ages

2. FOR counter FROM 1 TO 4 DO
3.         IF UserDetails [counter].ages > max
4.                 SET max TO UserDetails [counter].ages
5.         END IF
6. END FOR

7. SEND "The highest age is "& max TO DISPLAY
```

Count Occurrences

Count Occurrences is used to identify how many times a particular value appears in an array of record structure.

If used on the UserDetails record structure below for the name “Harry”, Count Occurrences would return the value, 2

RECORD *Userdetails* **IS**
{**STRING** Firstname, **STRING** Surname, **INTEGER** Age, **STRING** House}

First Name	Surname	Age	House
Harry	Jones	14	Bute
Jenna	White	12	Kintyre
Laura	Cairns	15	Arran
Sam	Kay	16	Arran
Harry	Smith	14	Lomond

```
1. RECEIVE target FROM KEYBOARD
2. SET numFound TO 0

3. FOR counter FROM 0 TO 4
4.      IF UserDetails [counter].names = target
5.              SET numFound TO numFound + 1
6.      END IF
7. END FOR

8. SEND “The number found is ”& numFound TO DISPLAY
```

Linear Search

Linear Search is used to identify whether or not an item is in a list, and which position it occupies.

If used on the UserDetails record structure below for the name, “Harry”, Linear Search would return the position, 4

RECORD *Userdetails* **IS**
{**STRING** Firstname, **STRING** Surname, **INTEGER** Age, **STRING** House}

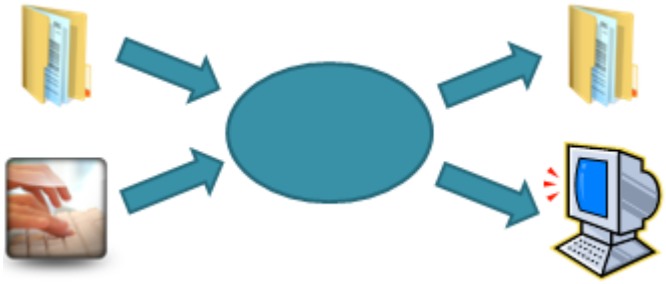
First Name	Surname	Age	House
Harry	Jones	14	Bute
Jenna	White	12	Kintyre
Laura	Cairns	15	Arran
Sam	Kay	16	Arran
Harry	Smith	14	Lomond

```
1. RECEIVE target FROM KEYBOARD
2. SET found TO FALSE
3. SET positionTO 0

4.   FOR counter FROM 0 TO 4
5.      IF UserDetails [counter].names = target
6.          SET found TO TRUE
7.          SET position TO counter
8.      END IF
9.   END FOR

10. IF found = TRUE
11.     SEND “Found at position ”& position TO DISPLAY
12. ELSE
13.     SEND “Not found”
14. END IF
```

Sequential File Operations

File handling is an alternative method to using the keyboard and display for input and output.

The purpose of file operations is to enable information to be received directly from a text file or sent directly to a text file.



Sending data to a file allows the output to be permanently stored.

The stored data could then be read back into the program the next time it is executed.

> *Imagine a computer game kept a high scores table.*
>
> *Without file handling, the high scores would only exist until the game was turned off. The next time you played, they would be gone.*
>
> *File handling allows the scores to be saved by the program and loaded in each time the game is started.*

Input from Sequential Files

If we wanted to input a score from a text file we would write:

```
OPEN "mytextfile.txt"

RECEIVE score FROM "mytextfile.txt"

CLOSE "mytextfile.txt"
```

Notice that the file has to be **opened** before it can be read and then **closed** again when the input is complete.

Output to Sequential Files

If we wanted to output a score to a text file we would write:

```
CREATE "mytextfile.txt"

OPEN "mytextfile.txt"

SEND score TO "mytextfile.txt"

CLOSE "mytextfile.txt"
```

Sequential File Operations

Open:	Initialises a file to prepare it to be read from or written to
Create:	Establish a new file and give it a name
Read:	Copy data from a file and store it in memory (variable/array)
Write:	Copy data memory (variable/array) and place it in a file
Close:	Close a file

Testing

Testing Stage

The Testing Stage is necessary in order to identify and correct and errors in the source code.

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**.

The test plan includes:

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

Test Plan

The test plan should ensure that testing is **comprehensive.**

Comprehensive testing is when the program is tested as thoroughly and completely as possible.

Ideally, ***exhaustive testing*** is used where every possible input and route through the program is tested – but this is not always practical or possible.

Comprehensive testing should involve using a range of **normal, extreme** and **exceptional** test data

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 and Debugging

There are three types of error that can occur when writing and testing a program.

Error Type	Description	Example
Syntax Error	The rules of the programming language have been broken. The program **will not** start.	FOR index **IS** 1 to 10 • is should be = IF age = 5 **THEEN** • 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 • Expected result is 10 but error in the first line means 24 is displayed.

Debugging is the process of finding and correcting errors.

Some types of error are easier to identify than others because the programming environment will help.

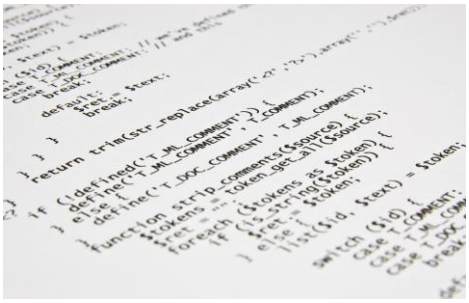
- The program will not run at all if there is a ***syntax* error**
- The program will stop running if an ***execution* error** is encountered

Logic errors are more difficult to find because the program will run but will produce incorrect results. There are a number of debugging techniques that can be used to identify logic errors.

- Dry Run
- Trace Table
- Trace Tools
- Breakpoints
- Watchpoints

Dry Run

A Dry Run involves **manually** stepping through each line of code using test data.



As lines of code that make changes to variables are reached, these changes are recorded using a **table**.

This should highlight positions in the code where variables are changing to unexpected values.

Trace Table

A trace table is similar to the table used to record variable values during a dry run.

Trace is often used to record the changes to variables when testing an algorithm for a specific sub-program.

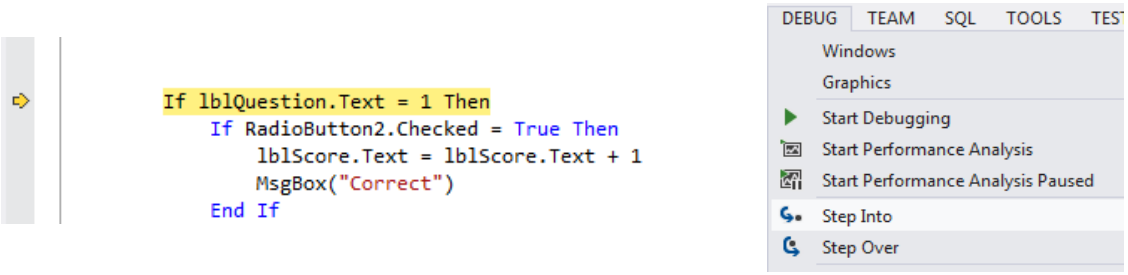
A trace table allows the tester to check the result of a number of different values of a variable.

	lower	upper	middle
1st pass			
2nd pass			

Trace Tools

Trace tools are a debugging feature of some programming environments.

Trace tools allow the program to be executed but the programmer can step through one line at a time.



This lets the programmer view the line of code being executed.

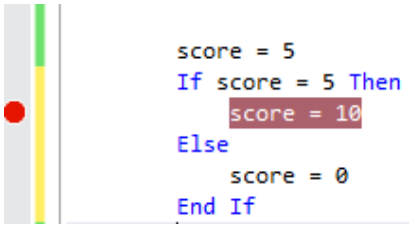
Breakpoints

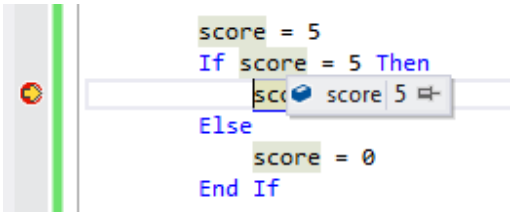
Breakpoints are another debugging feature of some programming environments.

Setting a breakpoint sets **a point in the code where the program will stop execution**.

Breakpoints are set to stop executing at a particular line of code.

Once the program has stopped, the values of variables can be examined and recorded in a trace table.

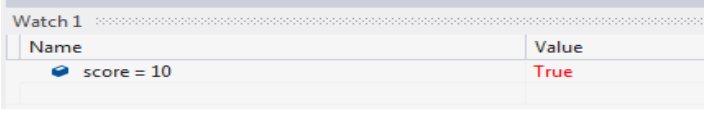


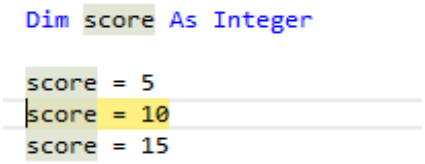


Watchpoint

A watchpoint is similar to a breakpoint but it does depend on reaching a particular line of code.

Instead, the program is set to stop executing when the value of a variable changes.





Again, once the program has stopped, the values of variables can be inspected and recorded in a trace table.

Evaluation

Evaluation Stage

During the Evaluation Stage, the overall success of the entire project is considered. This is an objective review of the software to establish whether it meets the required criteria.

An evaluation report would discuss:

Fitness for Purpose

This reflects whether the software carries out all the tasks required of the software specification.

Your evaluation should identify any discrepancies between the software specification and the completed software.

Efficient use of coding constructs

This reflects whether the software writers have used their knowledge of constructs to help them create efficient code. For example using:

- suitable data types or structures
- conditional or fixed loops
- arrays
- nested selection
- procedures or functions with parameter passing

Your evaluation should identify where your coding has been efficient.

Usability

This reflects how intuitive the software is from a user's perspective and should include:

- the general user interface
- the user prompts
- the screen layout
- any help screens

Your evaluation should identify features of the software that have enhanced usability for the user.

Maintainability

This reflects how easy it is to alter the software. The factors affecting maintainability include:

- readability of the code — made easier by using meaningful variable names, comments, indentation and whitespace
- amount of modularity — using functions and procedures effectively

Your evaluation should identify how your code helps with the maintainability of the software.

Robustness

This reflects how well the software copes with errors during execution including:

- exceptional data, e.g. the computer crashing if “out of range”
- incorrect data entered

Your evaluation should reflect the testing that has been undertaken to meet the specification, as well as to demonstrate some degree of robustness.