|  |  |
| --- | --- |
| badge 2.jpg | **Faculty of Business & Computing Science** |

**NAME \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |
| --- | --- |
| Computing Science |  |



Software Design and Development

**Computer Programming Notes**



Software Design and Development – Programming Notes

[1. Design Methodologies 5](#_Toc409609356)

[Rapid Application Development (RAD) 5](#_Toc409609357)

[Top-Down Design / Stepwise Refinement 6](#_Toc409609358)

[Agile Methodologies 7](#_Toc409609359)

[Waterfall Model of Software Development 8](#_Toc409609360)

[Personnel Involved 8](#_Toc409609361)

[Analysis Stage 9](#_Toc409609362)

[Design Stage 10](#_Toc409609363)

[Structured Diagrams 10](#_Toc409609364)

[Implementation Stage 11](#_Toc409609365)

[Testing Stage 11](#_Toc409609366)

[Documentation Stage 13](#_Toc409609367)

[User Guide 13](#_Toc409609368)

[Technical Guide 13](#_Toc409609369)

[Evaluation Stage 14](#_Toc409609370)

[Maintenance Stage 15](#_Toc409609371)

[2. Computer Languages 16](#_Toc409609372)

[Low Level Languages 16](#_Toc409609373)

[High Level Language 17](#_Toc409609374)

[Module Libraries 17](#_Toc409609375)

[Translators 18](#_Toc409609376)

[Interpreter 18](#_Toc409609377)

[Compiler 19](#_Toc409609378)

[Text Editor 19](#_Toc409609379)

[Types of Programming Languages 20](#_Toc409609380)

[Errors 21](#_Toc409609381)

[3. Programming Constructs – Visual Basic 22](#_Toc409609382)

[Storing Information 22](#_Toc409609383)

[Using Information 24](#_Toc409609384)

[Making Decisions – IF Statements 25](#_Toc409609385)

[Repeating Instructions – Loops 28](#_Toc409609386)

[Loops & Arrays Together 30](#_Toc409609387)

[Standard Algorithms 31](#_Toc409609388)

[Modular Programming 35](#_Toc409609389)

Software Design and Development – Outcomes

Outcome 1

1. **Explain how programs work, drawing on an understanding of advanced concepts in software development and computer architecture by:**

1.1 Reading and explaining code

1.2 Describing the purpose of a range of programming constructs and how they work

1.3 Describing how a range of standard algorithms work

1.4 Describing how programs relate to low-level structures and operations

Outcome 2

1. **Develop modular programs using one or more software development environments by:**

2.1 Applying contemporary design and development methodologies

2.2 Selecting and using combinations of appropriate constructs

2.3 Selecting and using appropriate simple and structured data types, including 1-D arrays

2.4 Testing digital solutions systematically

2.5 Identifying and rectifying program errors

2.6 Applying aspects of good programming technique — meaningful variable names, internal commentary, indentation

Outcome 3

1. **Produce a detailed report on the impact of contemporary computing technologies, by analysis and evaluating:**

3.1 Current trends in software development languages and environments

3.2 Current trends in the development of intelligent systems

3.3 Current trends in online systems

# Design Methodologies

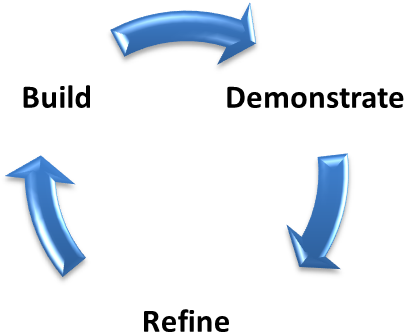
Software can be designed and developed by following different design methodologies. For this course you need to know about 4 of these.

* Rapid Application Development (RAD)
* Top-Down/Step-wise Refinement
* Agile Methodologies
* Waterfall Model

## Rapid Application Development (RAD)

Rapid application development (RAD) is a software development methodology that uses **minimal planning in favour of rapid prototyping**. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally **allows software to be written much faster, and makes it easier to change requirements.**

Analysis and Quick Design

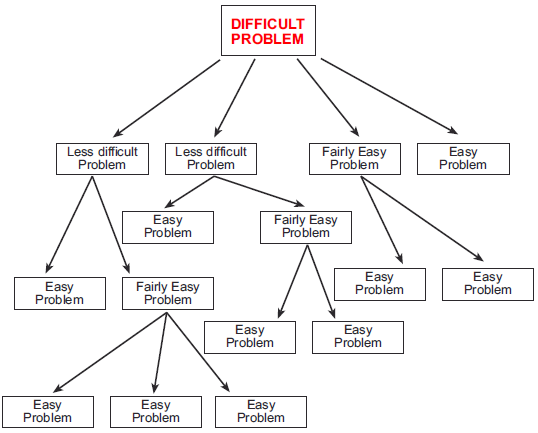


Testing

Implementation

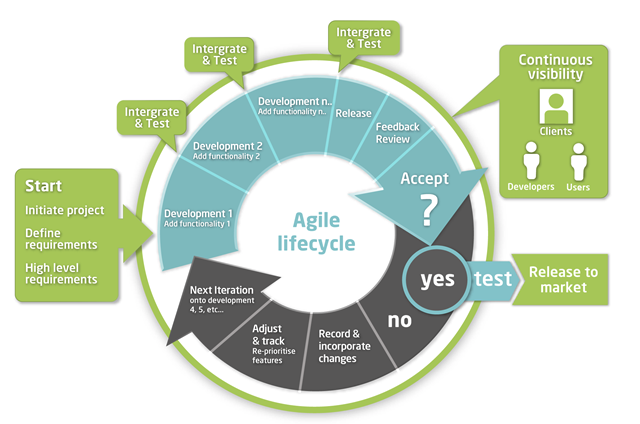
## Top-Down Design / Stepwise Refinement

|  |  |
| --- | --- |
| Top-Down Design | Step-Wise Refinement |
| This is where you start with a **problem at the top and work downwards in steps to smaller manageable problems** that are solvable. | This is the process of **breaking down large problems into smaller and smaller problems that are easier to solve**. It is easier to solve small simple problems than try and solve a large complex problem. |



These manageable parts can be split up and given out to different teams of programmers.

## Agile Methodologies

Agile methodologies are based on **iterative and incremental development**. Iterative means that stages of development can be revisited at any time and changes can be made. Throughout the development of a piece of software requirements and solutions evolve and change. Agile methodologies should be able to **respond to unpredictability and cope with rapid change** throughout the development cycle.

12 Agile Principles

The 12 Agile Principles are a set of guiding concepts that support project teams in implementing agile projects.

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

## Waterfall Model of Software Development

Good software is created as a result of well-documented stages. These stages are analysis, design, implementation, testing, documentation, evaluation and maintenance.

The software development process is known as an **iterative process**. This is because stages can be revisited as a result of information gained in later stages.

An error could be found in the testing stage and could cause the code to be changed and potentially alter the algorithm.

## Personnel Involved

Systems Analyst

The systems analyst **carries out analysis of a problem and acts as a communicator between the client and software development team**. The systems analyst has to be able to get a solid understanding of what the client wants and then communicate this to the development team.

Project Manager

The project manager is the person who is in charge of **keeping the whole project on schedule and within budget**. This person is also responsible for making sure the development team have the resources they need to be able to achieve the clients’ needs.

Client

The client represents **the management who require a new or updated piece of software**. The client has meetings with the development team to give them an idea of what their problem is.

Programmer

The programmer is **responsible for the coding, testing and maintenance of the software**. The programmer may be a part of team working on the same piece of software.

Independent Test Group

This is an **external group of people who will test the software** to find errors.

## 

## Analysis Stage

During this stage of development the **systems analyst analyses the existing system**. By getting an idea of the current system the systems analyst is able to understand what the client needs.

It is important that **the client gives a clear understanding of what they want**. This is will save time and money in the long run.

Software Specification

This is a document that is produced during the analysis stage that will **clearly identify the needs of the client.**

This document is a **legally binding contract between the developers and the client**. If the final product does not meet the requirements that the client asked for in the software specification then this document could be used in a court to support legal action.

Techniques used to extract information from the client

* Interviews
  + Employees could be interviewed to gain a better understanding of the current system and to identify any problems. From this they can build a clear picture of what needs to be done.
* Observation Notes
  + The everyday running of the business can be observed and notes made of what tasks people carry out in their role as part of the system.
* Questionnaires
  + Employees may be asked to complete a questionnaire that will give the development team a better understanding of the problem.

## Design Stage

Program designs can take many forms.

Pseudocode

A commonly used text based form of designing an algorithm for a program. One line of pseudocode normally translates to a line of program code. This form of design is written in English and should be understandable by most people.

Below is an example of some pseudocode that will store a patient’s heart rate each day.

Line 1 REPEAT

Line 2 RECEIVE bpm FROM keyboard

Line 3 IF bpm < 35 THEN

Line 4 SEND appropriate message TO display

Line 5 END IF

Line 6 UNTIL bpm >=35

**Structured Diagrams**

A form of **graphical design notation**. They show the hierarchy of the program components and how they are linked together. It should display the program in a series of smaller and smaller chunks. This form of design does not describe how the problem could be solved it focuses on how data should flow around the system.

## Implementation Stage

When the design is finished a programmer will then be given the task of **writing or implementing the code using a particular programming language**. There are many different programming languages for example Python, Visual Basic, C++, Java etc. The programmer will follow the design that was decided in the previous stage.

## http://bestoffshoreoutsourcing.com/wp-content/uploads/2012/02/Errors-in-Outsourcing.pngTesting Stage

Testing is very important to make sure the customer gets a program that is **error free and works under many different conditions**. Just like a product getting tested in many different forms so does software.

Comprehensive Testing

It is carefully planned to test a wide range of conditions. There are **three types of testing**. To show some test data under the headings. Let’s assume an exam was out of 100 marks.

Normal Test Data

* Making sure the program works when **used normally**. (An example of some test data could be: 21, 30, 76, and 80)

Extreme Test Data

* Making sure the program works when used that are **on the boundaries** of what could be considered normal. (An example of some test data could be: 0 and 100)

Exceptional Data

* Making sure the program can handle situations that it has **not been designed to cope with**. (An example of some test data could be: -1, 101, 78.008, 1000000000, abcde)

Beta Testing / Acceptance Testing

Many programs will be tested externally by a group of trusted users or the general public. This form of testing is known as **beta testing**. These trusted people will pass information back to the development team to make improvements and alterations to the program. This is also sometimes referred to as **acceptance testing**.

Systematic Testing

This type of testing involves going through a **progression from testing the sub-routines and working your way up to testing the entire system**. This sort of testing is **planned in advance and followed in a logical order**.

Breakpoints

The location in programming code that, when reached, triggers a temporary halt in the program. **Programmers use breakpoints to test and debug programs** by causing the program to **stop at specific lines of code so that the values of variables can be examined.**

Watchpoints

A watchpoint is when the programmer wants the code to stop when a specific condition is met. When this condition is met the programmer will examine the values of the variables,

Dry Runs / Trace Tables

In programming, a dry run is a **mental run of a computer program, where the programmer examines the code one step at a time** and determines what it will do when run. Dry runs are assisted with a table with the program or algorithm's variables on the top.

Advantages of using dry runs and trace tables

* Determine what code will do before it is run
* Spot logic errors in your code

## Documentation Stage

**User Guide**

When you buy a piece of software it comes with a piece of documentation called a user guide. This tells you **how to use the program**. A user guide normally contains a step-by-step tutorial taking you through the features and how to use them.

Some software comes with the user guide built into it on a CD/DVD or sometimes you can easily download it.

**Technical Guide**

This gives technical information such as the **system requirements** such as the amount of RAM and disk space needed to run the software. The system requirements should also state what operating systems are supported? The technical guide also includes instructions on **how to install the software**.



## Evaluation Stage

This is the last stage before the software is released. A report is done to evaluate the software and it should state whether or not the software is fit for purpose. The software is evaluated under the following headings:

Robust

The program should be able to cope with errors when the program is running.

Reliable

The program should work correctly if the correct data is entered.

Efficient

The program should be able to solve the problem without using too much memory and processing time

Portable

The program is easily adaptable to be run on different operating systems

Maintainable

The program should be able to have alterations made at a later date easily

Readable

The program should be easily understandable to another programmer. This is why it is important to use sensible variable names and include internal commentary.



## Maintenance Stage

This stage happens after the program has been released for use. There are three types of maintenance.

Corrective Maintenance

When programming large complex programs sometimes even after the testing stage errors and bugs slip through the net. This corrective maintenance stage involves **fixing them bugs that slipped through the testing stage by updating their app with a patch**. You may be familiar with apps on your smart phones getting updates when there has been a bug discovered.

Perfective Maintenance

You may notice that software normally has version numbers. Facebook App v5.0. These version numbers are due to **updates being made to the software that add new features to the software**. This may have been the result of users suggesting new features or from the evaluation stage.

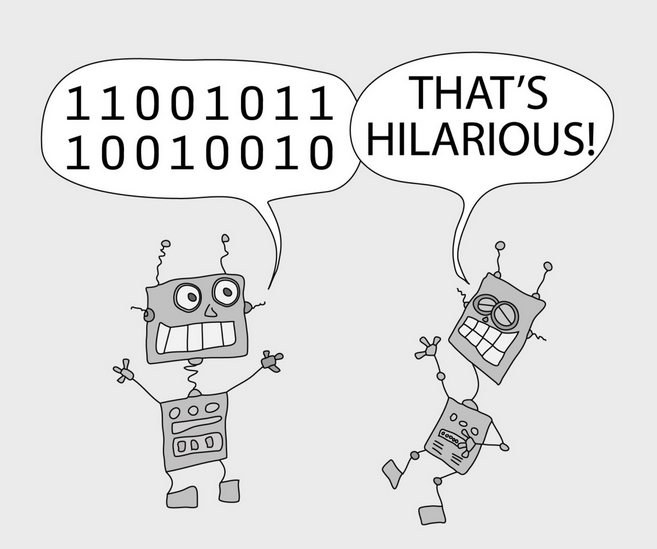
Adaptive Maintenance

This type of maintenance normally involves taking account of **new conditions such as the customer getting some new hardware or software**. Perhaps the customer updated their system to Windows 8 and the software you wrote no longer works. Adaptive maintenance will deal with updating your software to **work under new conditions such as a new operating system.**

# Computer Languages

Just like there are many languages spoken around the world, there are many computing languages used for programming. When you are in a foreign country you speak the appropriate language, similarly, you must use the appropriate programming language for the task you are completing.

## Low Level Languages

Inside every computer, there is a processor. This is a chip containing digital electronic circuits. These circuits work with tiny pulses of electricity and electronic components. The pulses of electricity can be **represented by the digits 1 and 0**. Every item of data and every instruction for the processor is represented by a group of these binary digits.

**Processors only 'understand' these binary digits**. The only inputs you can make to a processor are groups of binary digits. The only output that a processor can make is a group of binary digits.

Instructions and commands made for processors in this binary digital form are known as **machine codes**.

There are several problems with machine code:

* Machine codes are different for different processors
* They are very hard for humans to understand and use
* They take up a lot of space to write down
* It is difficult to spot errors in the codes.

Machine codes are an example of low-level language. To get around these difficulties computer scientists invented high-level languages.

## High Level Language

High level languages are **similar to human languages**. Instead of binary codes, they use **normal English words.**

With High Level Languages using English like words, it makes the code:

* easier to understand
* easier to spot errors
* more readable.

The following are examples of different high level languages:

|  |  |
| --- | --- |
| Name | Used for |
| PHP | general-purpose server-side scripting language |
| C++ | general-purpose programming language |
| Visual Basic | general-purpose interpreted high-level programming language |
| HTML | To create the structure of a web page |
| CSS | To control the style and look of a web page |

## Module Libraries

Module libraries contain a set of **pre-written and pre-tested subroutines** of code that are available to a programmer. The advantages of using these libraries are that you **don’t have to re-write code to problems already solved and the code is already checked for errors** so should be error free.

## Translators

High level languages have some great advantages compared to machine code. However, there is one major problem – processors don’t understand high level language at all. To solve this problem, computer scientists have developed translator programs that convert the high level language (written by humans) into machine code (understood by processors).

**High Level Language**

**Translator Program**

**Machine Code**

There are two types of translators that you need to be aware of:

* Interpreter
* Compiler

## Interpreter

An interpreter takes each line of high level language and translates it into machine code and passes it to the processor to carry out that instruction. It does this line by line. It will work through the entire program this way.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| A program will run even if it is not finished. | No copy of the machine code is saved. Meaning the source code has to be translated every time taking longer. |
| Easy to spot errors during the translation. | The process of translating the program slows down the running of it. |
| Program will run as soon as the first line is translated. | You will need to have a translator program or you cannot run it. |

## 

## Compiler

A compiler takes the high level language and translates the whole program into machine code once. The machine code can then be saved and kept and does not need to be translated before it is run. This is an example of .exe file. Software that you buy, such as a game or an application, will have been compiled into machine code before being distributed and sold.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| The machine code is saved so the program only needs to be translated once. | You have to wait until the code is complete and the errors have been fixed before the translation can be finished and the machine code is run. |
| The user does not need a translator program to run the machine code therefore the program runs quicker. | Each time the program is changed it needs to be re-translated. |

## Text Editor

A text editor is used to enter and alter source code. Text editors also have other useful features such as

* being able to copy and paste code and help complete a program statement
* formatting code
* indenting code
* adding line numbers

## Types of Programming Languages

There are many different programming languages, developed for different uses. You may already be familiar with some of these (Scratch, Visual Basic, HTML etc).

Within these different languages there are different types, again used depending on the requirements of the solution.

For this course, you need to know about three different types of languages.

* Procedural
* Declarative
* Object Orientated

Procedural Languages

Procedural languages are very popular in programming and most programmers are used to them. This type of language has a **clear start and end point**. The program will **follow a set pathway through the instructions to solve a problem**. Python is an example of a procedural language.

Declarative Languages

Declarative languages are based on **a collection of facts and rules that describe the problem**. The user would enter a query to question the knowledge base and return an answer. Prolog is an example of this. This type of language is geared more towards applications such as **artificial intelligence** where inexact data has to be handled or general decisions have to be made.

Object Orientated Languages (OOL)

Object Orientated languages **involve creating specific “objects”** that store data about each object. For example you could create an object to store information about you. It would contain data such as your name, address, phone number etc. Another example is creating an **object that would store car information such as registration number, drivers name, colour of car etc**. These objects are all represented by something called **classes**.

## http://bestoffshoreoutsourcing.com/wp-content/uploads/2012/02/Errors-in-Outsourcing.pngErrors

Even with practice, programmers will still make mistakes when writing code. There are different types of errors you should look out for:

Syntax Error

This is when your computer code is written incorrectly. As a result the compiler or the engine interpreting that code cannot understand what is going on. This can be from a bracket placed in the wrong spot or a misspelled word.

Execution Error

These are errors that occur once the program is running, rather than when being coded. For example, entering a letter when the program is expecting a number.

Logic Error

This is a bug in a program that causes it to operate incorrectly, but not to crash. A logic error produces unintended or undesired output or other behaviour, although it may not immediately be recognized as such.

For example when calculating average it should be.

average = (num1 + num2) / 2

**not**

average = num1 + num2 / 2

It is missing brackets in the calculation, so it compiles and runs but does not give the right answer. (Remember, maths is important in programming!)

# Programming Constructs – Visual Basic

The language that you are learning to write programs in is Visual Basic. You must be able to explain the following programming concepts and use them in a working program.

## Storing Information

When a computer program is running it needs a place to store information.

Variables – Definition (More on this later!)

* A variable is a data structure – a storage location within a program which allows for a single piece of data to be stored and then later retrieved.
* Variables are stored in RAM which means when the program stops running; the content of the variable is deleted.
* Variables must be given a variable
  + Names should be suitable (i.e. reflect what they are storing)
  + Names should not contain any spaces

Data

Variable

* You must declare the type of information that a variable can store:
  + String – stores characters, symbols and numbers.
  + Integer – stores positive and negative whole numbers.
  + Single – stores positive and negative real numbers (with decimal places).
  + Boolean – can be set to either true/false, yes/no.

|  |  |
| --- | --- |
| Syntax: | Dim **variable\_name** As **data\_type** |
| Example: | **Dim Name as String** |

Arrays

If a program needs to keep track of a related set of data, such as three people’s test marks, one approach would be to store each data item in a separate variable.

Test\_Mark\_1

Test\_Mark\_4

Test\_Mark\_3

Test\_Mark\_2

This can be time consuming to create and also limits what we can do with the information. Instead we can use an **Array**.

* An **array** is a data structure that can store more than one piece of information of the same type.

Test\_Mark(1)

* Each item in the list is referenced by giving each element in the list a unique number, called an **index**. This is the number in brackets.

Test\_Mark(2)

Test\_Mark(3)

* Just like a variable, an array must be given:
  + A suitable name

Test\_Mark(4)

* + A suitable data type
  + A suitable size

|  |  |
| --- | --- |
| Syntax: | Dim **array\_name** As **data\_type**(**array\_size)** |
| Example: | **Dim name As String(10)** |

## Using Information

Assignment

Having variables is one thing but the real use of variables is being able to change the value of the data that they store. This is made possible by being able to assign values to variables.

|  |  |
| --- | --- |
| Syntax: | **variable\_name = value** |
| Example: | **name = “Lauren”** |

In the above example, the variable **name** now contains the value “Lauren”. We can refer to this at any time in the program to use the information or change it.

Arithmetic

Programmers can perform arithmetic calculations using variables.

|  |  |
| --- | --- |
| **+** | Addition |
| **-** | Subtraction |
| **\*** | Multiplication |
| **/** | Division |
| **^** | Power |

|  |  |
| --- | --- |
| Example: | **Pay = 35 \* 8** |
|  | The result of 35 \* 8 (280) would be stored in the variable **Pay**. |

Arithmetic calculations can also be carried out using variables and not just to store the result.

|  |  |
| --- | --- |
| Example: | **Pay = hours\_worked \* hourly\_rate** |
|  | The value of **hours\_worked** is multiplied by the value of **hourly\_rate**. This answer is stored in the variable **pay**. |

## 

## Making Decisions – IF Statements

An IF Statement is used in program when a decision needs to be made. These are sometimes called selection statements. An IF statement will allow for a series of instructions to be executed based on the value of a condition.

|  |  |
| --- | --- |
| = | Equal to… |
| < | Less than… |
| > | Greater than… |
| <= | Less than or equal to… |
| >= | Greater than or equal to… |
| <> | Not equal to… |

Simple Conditions

* A simple condition is one which has only a **single** part to it.

For example, a theatre booking system might have to be written which applies a 10% discount if the theatre customer is under 18 years old. The program would need to know the age of the customer, and before it decided to apply a discount or not, it would have to check their age.

**age <= 18**

If the age is “less than or equal to” 18 then the condition is **true** and a discount is applied.

Complex Conditions

* A complex condition is one which has **two or more** parts to it.
* Complex conditions use logical operators including **AND, OR** or **NOT**.

For example, to know it’s summertime, a program could check whether the current month was June, July or August.

**Month = “June” OR Month = “July” OR Month = “August”**

Logical Operators

These are used to link parts of conditions together:

|  |  |
| --- | --- |
| AND: | Both parts of the condition must be true. |
| OR: | Either one **or** the other part of the condition must be true. |
| NOT: | Opposite. |

IF Statements

* An IF statement is used to make a decision.
* IF Statements work with either simple or complex conditions.

**IF *condition* THEN *action* ELSE(IF) *alternative action***

This is the question the program is asking.

If the answer is yes/true, carry out this action.

If the answer is no/false, carry out this action.

Almost all pieces of software require IF statements to work correctly. There are different types of IF statements:

|  |  |
| --- | --- |
| IF…THEN… | **IF grade = “A” Then**  **Msgbox(“Well Done! You got an A!”)**  **END IF** |
|  |  |
| IF…THEN…ELSE… | **IF grade = “A” Then**  **Msgbox(“Well Done! You got an A!”)**  **ELSE**  **Msgbox(“Sorry, you didn’t get an A.”)** |
|  |  |
| IF…THEN…ELSEIF… | **IF grade = “A” Then**  **Msgbox(“Well Done! You got an A!”)**  **ELSEIF grade = “B”**  **MSGBOX(“Well Done! You got a B!”)** |

If there are multiple possibilities then you can use a **CASE** statement.

|  |  |
| --- | --- |
| Case | **Select Case grade**  **Case “A”**  **Msgbox(“Excellent”)**  **Case “B”**  **Msgbox(“Good”)**  **Case “C”**  **Msgbox(“Satisfactory”)**  **Case “D”**  **Msgbox(“Must try harder.”)**  **End Select** |

## Repeating Instructions – Loops

In order to carry out a sequence of instructions a number of times, it is necessary to have **loops** in programming languages. These come in two distinct types, **fixed** and **conditional**.

Fixed Loops

* A fixed loop repeats a set of instructions a **fixed** number of times.
* The number of times is fixed at the start of the loop.

|  |  |
| --- | --- |
| **For…Next** Syntax | **For counter = 1 to *number of repetitions***  ***Instructions to be repeated***  **Next** |
|  |  |
| Example | **For counter = 1 to 10**  **Msgbox(“Hello”)**  **Next** |
|  | The above code will display the word “Hello” in a message box 10 times. |

There are a few points to remember:

* The **counter** is actually a variable which means you need to declare this at the start of your program.
* The **counter** variable keeps track of how many times the loop has been repeated (this can be very useful!)
* The number of repetitions can be a set number, or can be the value of a variable.
* A **nested loop** is a loop contained within another loop.

Conditional Loops

* A conditional loop is used when the number of repetitions is unknown.
* It will continue to repeat a set of instructions until a condition is met.
* Simple and Complex conditions can be used in a conditional loop.

|  |  |
| --- | --- |
| **Do…**  **Loop Until…**  Syntax | **Do**  ***Instructions to be repeated***  **Loop Until *condition to be met*** |
|  |  |
| Example | **Do**  **Order = Inputbox(“What is your order size?”)**  **Loop Until Order >= 1 AND Order <=100** |
|  | The above code will ask the user for their order size. It will keep asking them until they enter a number between 1 and 100. |
|  |  |
| **Do While…**  **Loop**  Syntax | **Do While*****condition***  ***Instructions to be repeated***  **Loop** |
|  |  |
| Example | **Word = Inputbox(“Enter word:”)**  **Do While (LEN(Word) > 5)**  **Msgbox(Word)**  **Loop** |
|  | The above code will ask the user for a word. As long as that word is greater than 5 characters long it will be displayed in a message box. |

## Loops & Arrays Together

Large complicated programs involve bringing all the concepts you have learned and making them work together. One example of this is using an **array** within a **loop**.

For example, a teacher wants to store the names of 5 pupils and their prelim test score. He could use 5 different variables for each of the names, and another 5 different variables for each of the scores. However, this causes problems because the mark does not link to the name. The solution is to use **arrays** and a **loop**.

Have a look at the code:

The 5 indicates the size of the array.

**counter** keeps track of the number of repetitions i.e. first time, counter = 1

second time, counter = 2

|  |
| --- |
| **Dim name(5) As String**  **Dim score(5) As Integer**  **Dim counter As Integer**  **For counter = 1 to 5**  **name(counter) = Inputbox(“Enter name:”)**  **score(counter) = Inputbox(“Enter the score for & name(counter))**  The first time round the loop, the name is stored in location one. The second time round the loop, the name is stored in location two.  This also happens for score.  **Next**  **For counter = 1 to 5**  **listbox.Items.Add(name(counter))**  **listbox.Items.Add(name(counter))**  **Next** |

This loop displays each item in the array in a list box, one by one.

## Standard Algorithms

A standard algorithm is a set of programming instructions – to achieve a specific task. These programming instructions follow a standard pattern which means they can be used in different situations. This standard pattern saves a programmer time when designing programs.

Input Validation

* This standard algorithm checks an input to see if its value lies within a given range.
* If the value is within the range, it is accepted as valid.
* If the value is not within the range, an error message is displayed and the user will need to re-enter the value.
* The program will continue to ask for value until it is within the given range.

|  |  |
| --- | --- |
| Pseudocode | 1. **Start Loop** 2. **Get input from user** 3. **Check if input is valid** 4. **If not, give error message** 5. **Repeat until input is valid** |
|  |  |
| Example | **Do**  **number = Inputbox(“Please enter a number between 1 & 100”)**  **IF number < 1 or number > 100 THEN**  **MSGBOX(“Please enter a number between 1 & 100”)**  **Loop Until number >= 1 and number <=100** |
|  | The above code will ask the user for a number between 1 and 100. If the number is not in this range, the user will see an error message and be asked to enter it again. The program will keep asking the user until a number between 1 and 100 is entered. |

You should notice that Input Validation uses a number of different programming techniques:

* Assignment
* IF statements
* Conditional Loops

Finding Maximum/Minimum

These algorithms can be used to find the **highest** or **lowest** value in a list.

|  |  |
| --- | --- |
| Pseudocode | 1. **Set MAX to first item in the list** 2. **Loop for each item in the list** 3. **IF current item is > than MAX THEN** 4. **MAX becomes the current item** 5. **End if** 6. **End Loop** |
|  |  |
| Example | **max = list(1)**  **For index = 1 to 100**  **IF list(index) > max THEN**  **max = list(index)**  **END IF**  **Next** |
|  | This code will look for the highest value in a list. The highest value is set to the first item in the list. It will then check every item in the list and if the current value is greater than the max, it is replaced with this value. |

Finding minimum requires only minor changes:

|  |  |
| --- | --- |
| Pseudocode | 1. **Set MIN to first item in the list** 2. **Loop for each item in the list** 3. **IF current item is < than MIN THEN** 4. **MIN becomes the current item** 5. **End if** 6. **End Loop** |

Linear Search

This algorithm will look at a list of items and search for a “target value".

It must look at each item in the list and compare the item to the “target value”. If the current item matches the “target value”, then the program reports that a match has been found.

Consider the example of looking for a name in a list of 100 names:

|  |  |
| --- | --- |
| Pseudocode | 1. **Get target value from user** 2. **Set found to false** 3. **Loop for each item in the list** 4. **IF current item matches target value** 5. **Set found to true** 6. **Set position = current position** 7. **End Loop** 8. *If found = true then* 9. *Send found message to display* 10. *Else* 11. *Send not found message to display* 12. *End if* |
|  |  |
| Example | **target = Inputbox(“Enter name”)**  **found = false**  **For index = 1 to 100**  **IF name(index) = target THEN**  **Found = true**  **position = index**  **END IF**  **Next**  **If found = true then**  **Msgbox(“ The name was found “)**  **Else**  **Msgbox(“ The name was not found “)**  **End if** |
|  | The above code will ask the user for a target name to search for. The program will look at each item in the list and if the current name matches the target name, the program will store the list position of that name.  *Note that if the program finds more than one matching name, it will only remember the position of the last match.* |

Counting Occurrences

This algorithm uses a variable together with a suitable type of loop to find out how many times some condition has been met. For example, how often a word appears in a list.

It might be suitable to use a fixed loop, a pre-conditional loop, or a post-conditional loop, depending on the circumstances.

Consider the example of counting how many of the 30 possible students got at least 70 marks out of 100 in a Computing test:

|  |  |
| --- | --- |
| Pseudocode | 1. **Set mark counter to 0** 2. **Loop for each item in the list** 3. **If current mark => 70 then** 4. **Increment the mark counter** 5. **End if** 6. **End the loop** 7. **Send message to display** |
|  |  |
| Example | **mark\_count = 0**  **For index = 1 to 30**  **IF marks(index) => 70 THEN**  **mark\_count = mark\_count + 1**  **END IF**  **Next**  **Msgbox (“The number of pupils who achieved 70 or more marks was “& mark\_count)** |
|  | The above code will look at each item in the list. If the current item in the list is greater than or equal to 70 then the mark counter will be increased by 1. |

## Modular Programming

All good programs will have a very clear structure. The best way to achieve this is to break programs down into smaller chunks. These are known as **subroutines**.

**Subroutine** – a block of code that does a specific job.

Programs that make use of subroutines are known as **modular programs**. There are benefits to using modular programs:

* Each of the subroutines can be used in any order and can be reused multiple times
* Different teams of programmers can work on different subroutines at the same time
* The same block of code can be reused to perform slightly different jobs by passing in different parameters (we’ll look at this later).

There are **3** types of subroutines that you need to be aware of:

* Procedures
* Functions
* Methods

These different subroutines can be used at different times in a program. To use one of these subroutines we use the term **call**.

|  |  |
| --- | --- |
| Procedures | Functions |
| A section of code to perform a specific task with the information provided to it  (parameter passing). | |
| Does not return a value to the main program. | **Does** return a value to the main program. |

In visual basic, you need to be able to program with procedures and functions but not methods.

Parameter Passing

This is the term used for passing information to and from a subroutine. A **parameter** is a value that is being passed in or out of a block of code, i.e. a subroutine. There are two methods of passing a parameter.

|  |  |
| --- | --- |
| By Reference | By Value |
| Used when a value is being passed into a subroutine. | Used when a value is being passed **out** of a subroutine |
| The original data is changed by the subroutine. | The original data **is not changed** by the subroutine. |
| Arrays are always passed by reference. |  |

Variables – Score

With modular programs we must consider the **scope** of variables. Scope means what parts of the program can access the variable.

|  |  |
| --- | --- |
| Local Variable | Global Variable |
| This is a variable that is only used within a single block of code. | A global variable is created in the main part of the program. |
| It cannot be seen or accessed from any other part of the program. | It can be seen and accessed from all parts of the program. |

* Using global variables can result in data being changed by more than one subprogram, which can result in unexpected results from the program.
* Using global variables means that programmers on the entire project need to ensure that they choose variable names that are unique across the whole project.
* Global variables can make it more difficult to reuse sections of code, as the code cannot simply be copied from one program to another without making sure that all the relevant global variables are also copied.