

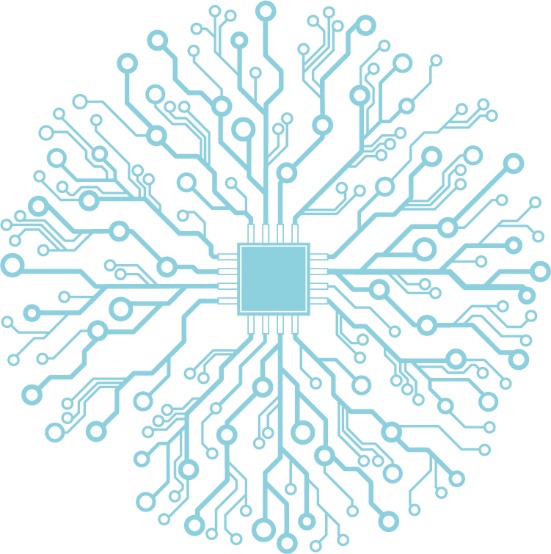
# J277 Guide to programming techniques: Visual Basic .NET

### Introduction

This guide is designed to support candidates’ learning about how to use Visual Basic .NET and how it relates to the OCR Exam Reference Language.

Please refer to the [J277 Specification](https://www.ocr.org.uk/Images/558027-specification-gcse-computer-science-j277.pdf), Section 2.2 for a full list of skills/techniques that candidates must be familiar with.

*Disclaimer: Please note that this is not a complete guide to Visual Basic .NET and only explores some of the ways to use Visual Basic .NET to express the techniques in the specification.*



# Contents

[J277 Guide to programming techniques: Visual Basic .NET 1](#_Toc77318917)

[Introduction 1](#_Toc77318918)

[Contents 2](#_Toc77318919)

[The use of variables 5](#_Toc77318920)

[Key words for Visual Basic .NET data types 6](#_Toc77318921)

[Variable naming 7](#_Toc77318922)

[Constants 7](#_Toc77318923)

[Operators 8](#_Toc77318924)

[Relational/equality operators 8](#_Toc77318925)

[Arithmetic operators 9](#_Toc77318926)

[Logical operators 10](#_Toc77318927)

[Precedence 11](#_Toc77318928)

[Inputs 11](#_Toc77318929)

[Entering strings/text 11](#_Toc77318930)

[Entering numbers 12](#_Toc77318931)

[Outputs and assignments 13](#_Toc77318932)

[Sequence 13](#_Toc77318933)

[Selection 14](#_Toc77318934)

[IF/ELSE or SWITCH/CASE 15](#_Toc77318935)

[ELSEIF statements 15](#_Toc77318936)

[Logic errors in IF statements 16](#_Toc77318937)

[Nesting IF statements 16](#_Toc77318938)

[Iteration 17](#_Toc77318939)

[FOR Loops/Count-Controlled 17](#_Toc77318940)

[Counting forwards 18](#_Toc77318941)

[Counting backwards 19](#_Toc77318942)

[Condition-controlled loops 19](#_Toc77318943)

[WHILE loops 20](#_Toc77318944)

[DO UNTIL loops 21](#_Toc77318945)

[**Infinite loops** 22](#_Toc77318946)

[Using break and continue key words 22](#_Toc77318947)

[***The use of the break in loops*** 22](#_Toc77318948)

[***The use of the continue keyword*** 22](#_Toc77318949)

[Examples of break and continue in loops 23](#_Toc77318950)

[The use of basic string manipulation 24](#_Toc77318951)

[Placeholders 25](#_Toc77318952)

[‘Contains’ within a string 26](#_Toc77318953)

[ASCII values 26](#_Toc77318954)

[Substrings 27](#_Toc77318955)

[String cutting from the left 27](#_Toc77318956)

[String cutting from the right 27](#_Toc77318957)

[File handling 27](#_Toc77318958)

[The use of close 28](#_Toc77318959)

[Reading a file line by line 29](#_Toc77318960)

[Reading a whole file at once 29](#_Toc77318961)

[Write to a file (overwrite) 30](#_Toc77318962)

[Write to a file (append) 30](#_Toc77318963)

[The use of arrays 31](#_Toc77318964)

[Declaring an array 31](#_Toc77318965)

[Initialising an array with data items 32](#_Toc77318966)

[Assigning data items to an array after initialisation 32](#_Toc77318967)

[Sorting arrays 33](#_Toc77318968)

[Reverse sorting arrays 33](#_Toc77318969)

[Printing arrays 35](#_Toc77318970)

[Using a FOR loop 35](#_Toc77318971)

[Using a FOR EACH loop 36](#_Toc77318972)

[The use of records to store data 37](#_Toc77318973)

[Printing a record structure 38](#_Toc77318974)

[How to use sub programs 39](#_Toc77318975)

[Creating a procedure 40](#_Toc77318976)

[Creating a function 40](#_Toc77318977)

[Sub programs with parameters 40](#_Toc77318978)

[Single parameter functions 40](#_Toc77318979)

[Multiple parameter functions 40](#_Toc77318980)

[Using sub programs within a program 41](#_Toc77318981)

[Using returned values 41](#_Toc77318982)

[Where should I use functions? 42](#_Toc77318983)

[Casting 43](#_Toc77318984)

[Random numbers 44](#_Toc77318985)

[Combinations of techniques 45](#_Toc77318986)

[Recursion 47](#_Toc77318987)

[Scope 48](#_Toc77318988)

[Local variables 48](#_Toc77318989)

[Avoiding scope issues 49](#_Toc77318990)

[**Global variables** 49](#_Toc77318991)

[The use of SQL to search for data 50](#_Toc77318992)

**OCR Resources**: *the small print*OCR’s resources are provided to support the delivery of OCR qualifications, but in no way constitute an endorsed teaching method that is required by the Board, and the decision to use them lies with the individual teacher. Whilst every effort is made to ensure the accuracy of the content, OCR cannot be held responsible for any errors or omissions within these resources.

Our documents are updated over time. Whilst every effort is made to check all documents, there may be contradictions between published support and the specification, therefore please use the information on the latest specification at all times. Where changes are made to specifications these will be indicated within the document, there will be a new version number indicated, and a summary of the changes. If you do notice a discrepancy between the specification and a resource please contact us at:

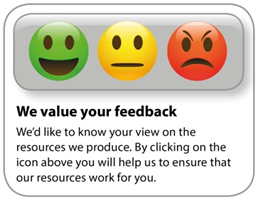
[resources.feedback@ocr.org.uk](file:///\\filestorage\OCR\PD\ProdSup\Design\Studio\Visual%20Style%20Guidelines\2016_Templates\resources.feedback@ocr.org.uk).

© OCR 2020 - This resource may be freely copied and distributed, as long as the OCR logo and this message remain intact and OCR is acknowledged as the originator of this work. OCR acknowledges the use of the following content: n/a

Please get in touch if you want to discuss the accessibility of resources we offer to support delivery of our qualifications: [resources.feedback@ocr.org.uk](mailto:resources.feedback@ocr.org.uk)

Whether you already offer OCR qualifications, are new to OCR, or are considering switching from your current provider/awarding organisation, you can request more information by completing the Expression of Interest form which can be found here: [www.ocr.org.uk/expression-of-interest](http://www.ocr.org.uk/expression-of-interest)

Looking for a resource? There is now a quick and easy search tool to help find free resources for your qualification:   
[www.ocr.org.uk/i-want-to/find-resources/](http://www.ocr.org.uk/i-want-to/find-resources/)



# The use of variables

**OCR Exam Reference Language**

| x = 3  name = "Bob" | Variables are assigned using the = operator. |
| --- | --- |
| const pi = 3.14 | Variables in the main program can be made a constant with the keyword const. |
| global userID = 123 | Variables in the main program can be made global with the keyword global. |

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim parrotAge  dim parrotAge as integer  dim parrotAge = 10  dim parrotAge as integer = 10  const parrotStatus = "Alive" | Visual Basic .NET requires you to declare the variable using the dim keyword (abbreviation of dimension) However, it does not require you to assign a value or data type to the variable straight away.If a data type is omitted the variable will be assigned a data type based on the first value it is assigned.  We can use the keyword const to ensure that parrotStatus cannot be changed during runtime  There is no global key word in Visual Basic .NET. The scope of a variable or constant is determined by where it is declared. This is discussed further in the ‘Scope’ section of the guide. |
| dim parrotAge = 10  const parrotStatus = "Alive"  parrotAge = 12  Console.WriteLine("The parrot is currently " & parrotAge & " and is " & parrotStatus) | Once declared you can use the variable with other values or variables as shown. The ‘&’ is used to join values together.  Note that if you leave the variable ‘null’ when you declare it, you must assign a value to the variable first before being able to carry out further operations on it. Screen shot |
| parrotAge = parrotAge + 1  Console.WriteLine(parrotAge) | A variable can be overwritten with a new value at any time. |
| Screen shot | You cannot assign data to a variable of different types. Each variable will only hold the data type defined.  As you can see – an error is shown in the IDE as "two" is a string, and we are trying to assign it to an ‘int’. |

### Key words for Visual Basic .NET data types

|  |  |
| --- | --- |
| **Data Type** | **Range** |
| **byte** | 0 .. 255 |
| **sbyte** | -128 .. 127 |
| **short** | -32,768 .. 32,767 |
| **ushort** | 0 .. 65,535 |
| **integer** | -2,147,483,648 .. 2,147,483,647 |
| **uinteger** | 0 .. 4,294,967,295 |
| **long** | -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807 |
| **ulong** | 0 .. 18,446,744,073,709,551,615 |
| **double** | -1.79769313486232e308 .. 1.79769313486232e308 |
| **decimal** | -79228162514264337593543950335 .. 79228162514264337593543950335 |
| **char** | A Unicode character. |
| **string** | A string of Unicode characters. |
| **boolean** | True or False. |

### Variable naming

There are some basic rules with variable names in Visual Basic .NET

* the first character of a variable must be either a letter or an underscore (\_).
* after this, they can be any combination of letters, underscores or numbers.
* they can only use letters, numbers and underscores (\_)
* hyphens are not allowed (-)
* spaces are not allowed
* they can’t begin with a number
* special characters are not allowed such as $ or ‘.

Remember:

* variable names are not case sensitive, SPAM and spam are the same variables
* it is convention to use a lower case letter at the start of a variable name
* you can use camelCase or not\_camel\_case
* a good variable name describes the data it contains
* the variable type comes after the variable name in Visual Basic .NET.

# Constants

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| const pi = 3.14 | Variables in the main program can be made a constant with the keyword const. |

**Visual Basic .NET**

|  |  |
| --- | --- |
| const ParrotAge = 0  const ParrotAge as integer = 0 | The keyword for declaring a constant is **const**. It is used BEFORE declaration of the variable and the data type is optional.  If you try to use the keyword **const** and do not declare a value to the variable, then it will throw an error! As you can see – there is a warning as parrotAge is underlined.  **Remember a constant cannot have a new value assigned to it during run time.** |

# Operators

### Relational/equality operators

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

When using Logical Operators, the answer to a comparison is always TRUE or FALSE (i.e. a Boolean result. Have a look at what happens with the following code, and results:

|  |  |
| --- | --- |
| dim valueA = 23  dim valueB = 15  Console.WriteLine(valueA = valueB)  Console.WriteLine(valueA <> valueB)  Console.WriteLine(valueA < valueB)  Console.WriteLine(valueA <= valueB)  Console.WriteLine(valueA > valueB)  Console.WriteLine(valueA >= valueB) | Screen shot |

You can also save these results as a variable.

|  |  |
| --- | --- |
| dim valueA = 23  dim valueB = 15  dim myResult = false  myResult = valueA <> valueB  Console.WriteLine("My Result = " & myResult) | Screenshot |

### Arithmetic operators

|  |  |
| --- | --- |
| + | Addition e.g. x=6+5 gives 11 |
| - | Subtraction e.g. x=6-5 gives 1 |
| \* | Multiplication e.g. x=12\*2 gives 24 |
| / | Floating point division e.g. x=13/2 gives 6.5 |
| \ | Integer division e.g. x=13\2 gives 6 (DIV operator) |
| MOD | Modulus (integer remainder) e.g. 12 MOD 5 gives 2 |
| ^ | Exponentiation e.g. 3^4 gives 81 |

Examples of Arithmetic operators:

|  |  |
| --- | --- |
| dim valueA = 23  dim valueB = 15  Console.WriteLine(valueA + valueB)  Console.WriteLine(valueA - valueB)  Console.WriteLine(valueA \* valueB)  Console.WriteLine(valueA \ valueB)  Console.WriteLine(valueA MOD valueB) | Screenshot |
| dim number1 = 2  dim number2 = 2  dim result = number1 ^ number2  Console.WriteLine(number1 & " ^ " & number2 & " = " & result) | Screenshot |

### Logical operators

* AND
* OR

Logical operators chain relational operators together, where you may need two or more things to be either TRUE or FALSE to continue.

|  |
| --- |
| dim caster = "witch"  dim victim = "peasant"  dim later = True  dim spellCast = "She turned me into a newt"  dim victimStatus = "I got better!"  if ((caster = "witch") AND (victim = "peasant")) then  Console.WriteLine(spellCast)  end if |
| Screen shot- Both conditions here evaluate to true  - Therefore the IF statement condition evaluates to true  - Thus the IF statement is executed |

|  |  |
| --- | --- |
| dim caster = "witch"  dim victim = "peasant"  dim later = True  dim spellCast = "She turned me into a newt"  dim victimStatus = "I got better!"  if ((caster = "witch") and (victim = "dog")) then  Console.WriteLine(spellCast)  end if | Here the IF statement will not execute because the victim is not a dog.  BOTH conditions must be true for the IF statement to execute. |
| dim caster = "witch"  dim victim = "peasant"  dim later = True  dim lie = True  dim spellCast = "She turned me into a newt"  dim vicitmLies = "We believe you are lying!"  dim victimStatus = "I got better!"  if ((caster <> "witch") OR (lie = True)) then  Console.WriteLine(vicitmLies)  end if | Screen shot   * At least one condition is true * Therefore the IF statement condition evaluates to **true** * Thus the IF statement is executed |

### Precedence

Visual Basic .NET uses standard orders or precedence. When using long calculations, remember that using brackets can change the results significantly!

|  |  |
| --- | --- |
| dim valueA = 23  dim valueB = 15  dim valueC = 15  dim valueD = 4  dim resultA, resultB  resultA = valueA + valueB \ valueC - valueD  resultB = (valueA + valueB) \ (valueC - valueD)  Console.WriteLine("Result A = " & resultA)  Console.WriteLine("Result B = " & resultB) | Screenshot  - Note that adding brackets has changed the result.  - Remember BODMAS (or BIDMAS!) |

# Inputs

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| myName = input("Enter name") | Here we declare a variable myName and assign the input to it. We also prompt the user as to what to input. |

**Visual Basic .NET**

Everything a user enters at the console is treated as text. Visual Basic .NET will not convert say a number you enter into an integer, or a double. You need to tell it to do that. Therefore there are two ways you can enter data into Visual Basic .NET:

### Entering strings/text

|  |  |
| --- | --- |
| dim myName = ""  Console.Write("Please enter your name: ")  myName = Console.ReadLine()  Console.WriteLine("Your Name: " & myName) | Screen shot |

### Entering numbers

Numbers need to be converted (or cast) on entry. You can cast to any numerical data type. Remember to try and keep the variable data type the same as the data type you are assigning to! If you do not, the compiler **may** not do what you intended.

|  |  |
| --- | --- |
| dim valueA = 23  Console.Write("Please Enter an Integer: ")  valueA = cint(Console.ReadLine())  Console.WriteLine("Your Value: " & cstr(valueA)) | Screen shot |

**Casting a decimal to be stored as an integer will be rounded to the nearest value:**

|  |  |
| --- | --- |
| dim valueA = 23  Console.Write("Please Enter a Decimal: ")  valueA = cdec(Console.ReadLine())  Console.WriteLine("Your Value: " & cstr(valueA)) |  |

**Casting an integer to be stored as a decimal WOULD work:**

|  |  |
| --- | --- |
| dim valueA = 23.0  Console.Write("Please Enter an Integer: ")  valueA = cint(Console.ReadLine())  Console.WriteLine("Your Value: " & cstr(valueA)) | Screenshot |

# Outputs and assignments

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| print(string)  print(variable) | Outputs the argument (string or variable) to the screen. |

**Visual Basic .NET**

|  |  |
| --- | --- |
| Visual Basic .NET uses a library to input and output to console. It should be imported already, when you create a new project within your IDE.  Without the highlighted text (right) the System Libraries would not be imported, and therefore you could not use the Console.WriteLine() method. | Imports System |
| Console.WriteLine("How do you know she's a witch?") | |
| Here the Console.WriteLine method takes the string and outputs to the screen. | |
| dim myAnswer = "She turned me into a newt!"  Console.WriteLine("How do you know she's a witch?")  Console.WriteLine(myAnswer) | |
| Screenshot | |

### Sequence

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| x = 3  y = 2  x = x + y  print(x) | x is assigned the value of 3, y is assigned the value of 2. x is then re-assigned to be the value of 3 plus 2 which evaluates to 5 and is printed to the screen.  It should be noted that that value of x changes in sequence, line by line as it is interpreted, at the start of line 3 (x=x+y) x still has a value of 3 but once that line is run it then changes to be x+y or 5. |

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim x = 3  dim y = 2  x = x + y  Console.WriteLine(x) | Screenshot |

### Selection

It helps to think of selection as a test of a condition such as:

**if** some condition is met **then**

**do** something

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| if entry == "a" then  print("You selected A")  elseif entry == "b" then  print("You selected B")  else  print("Unrecognised selection")  endif  switch entry:  case "A":  print("You selected A")  case "B":  print("You selected B")  default:  print("Unrecognised")  endswitch | Selection will be carried out with if/else and switch/case. In the example the OCR Exam Reference Language is checking the input and returning a message based upon the specific input required, the else block is used as a catch for any unexpected input which allows the code to degrade gracefully.  The switch/case is slightly different in that you always have to have the ‘default’ case.  This effectively creates a ‘Catch’ for when invalid entry is recognized. In an IF statement you would have to specifically create an ELSE construct to cope for this. |

### IF/ELSE or SWITCH/CASE

IF statements use relational operators to make decisions. Is a variable equal to that variable, is it greater than etc.

Case Selects only look for exact matches, via the ‘case’ line. Therefore, you have to make a decision as to what choice you are making, as to which may be most appropriate.

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim duckWeight = 15  dim personWeight = 13  if (duckWeight >= personWeight) then  Console.WriteLine("Clearly not a witch!")  else  Console.WriteLine("She's a Witch!")  end if | Screenshot  Here the weight of the person is not greater than that of the duck, and therefore the IF statement executes, and the ELSE is skipped. |
| dim duckWeight = 15  dim personWeight = 34  if (duckWeight >= personWeight) then  Console.WriteLine("Clearly not a witch!")  else  Console.WriteLine("She's a Witch!")  end if | Screen shot  Here the weight of the person is greater than the duck. This means that the IF statement is FALSE and does not run.  Therefore, the ELSE statement executes by default. |

### ELSEIF statements

|  |  |
| --- | --- |
| dim duckWeight = 15  dim personWeight = 15  if (duckWeight > personWeight) then  Console.WriteLine("Clearly not a witch!")  elseif (personWeight > duckWeight) then  Console.WriteLine("She's a Witch!")  else  Console.WriteLine("They weigh the same!")  end if | Screen shot  Note the subtle change in logic in the first IF statement with the removal of the ‘=’.  Now the first two tests are for greater than. As the weights are the same, it checks both IF statements, which return false, and therefore executes the ELSE statement. |

### Logic errors in IF statements

|  |  |
| --- | --- |
| dim duckWeight = 15  if (duckWeight > 5) then  Console.WriteLine("The duck is a small duck!")  elseif (duckWeight > 10) then  Console.WriteLine("The duck is a normal size!")  elseif (duckWeight >= 15) then  Console.WriteLine("The duck is a large duck!")  end if | When using logic statements you need to be careful about the order you carry out your IF statement conditions.  Here, you can see that the duckWeight is 15. By looking at the IF statement results, we can see that the duck SHOULD be reported as being a small duck. (yellow highlight)  However, the first IF condition checks for duckWeight being greater than 5. As 15 > 5 = True, it will use the FIRST IF statement, and then skip the rest. |

### Nesting IF statements

It is possible to **nest** IF statements.

|  |  |
| --- | --- |
| if (*Case A is true*) then  'This code will execute  if (*Case B is true*) then    'This nested IF code will execute    end if  else  Console.WriteLine("They weigh the same!");  end if | Note the change in logic.  If Case A is **false**, the ELSE statement will execute.  If Case A is **true** then any code in that IF statement will run.  This will check the NESTED **if** statement.  Case B will ONLY be checked **if** Case A is true.  Case B IF statement code will ONLY execute if **both** Case A and Case B are true. |

# Iteration

### FOR Loops/Count-Controlled

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| for count=0 to 7  print ("Hello")  next i  for count=0 to 7 step 2  print ("Hello")  next i | Will print "Hello" 8 times (0-7 inclusive). Note that the count starts at 0. |

**Visual Basic .NET**

Count controlled loops will need a variable to be incremented or decremented to define when the loop ends.

The structure of the FOR Loop is:

for counter = startValue to endValue step incrementValue

next counter

The variable (counter) you use to ‘control’ iterations with can either be created inside the loop itself, or use a variable that already exists within the program.

The counter operation always executes at the end of each iteration.

|  |  |  |
| --- | --- | --- |
| **Counter created** | **Benefits** | **Drawbacks** |
| In the loop | * + - Easy to see where the loop terminates | * + - Loop will always have fixed iterations |
| Using pre-existing variable | * + - Allows you to link the loop to other variables in your program     - Allows varying number of iterations to take place | * + - Can lead to run time issues |

### Counting forwards

|  |  |
| --- | --- |
| for count = 1 to 5  Console.WriteLine(count)  next count | Here is an example of the FOR loop in action.  Our count is defined within the loop itself. We always know this loop will start a 1 and then iterate until the count = 5. When the count ‘i’ reaches 6, the loop will exit as the condition becomes false.  Screen shot |
| dim countTo = 0  Console.Write("What number do you want to count to?: ")  countTo = integer.Parse(Console.ReadLine())  for count = 1 to countTo  Console.WriteLine(count)  next count | Here we are using a variable within the condition to vary the number of iterations.  This means the loop has more functionality:  Run 1:  Screen shot  Run 2:  Screenshot  However, what would happen if the user entered ‘0’ for a number to count to? |

### Counting backwards

|  |  |
| --- | --- |
| dim countFrom = 0  Console.Write("What number do you want to count from?: ")  countFrom = integer.Parse(Console.ReadLine())  for count = countFrom to 0 step -1  Console.WriteLine(count)  next count | You can also decrease your counter to reach a certain value.  Screesnshot |

### Condition-controlled loops

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| while answer != "Correct"  answer = input("New answer")  endwhile | Condition-controlled loop, this will loop until the user inputs "Correct". It will check the condition before entering the loop. |
| do  answer = input("New answer")  until answer != "Correct" | The loop iterates once before the check is carried out.  Note that the 'until' means that the logic of the loop has now changed. **Be careful** when writing this in Pseudocode! |

### WHILE loops

Where we do not know when the end of the loop may occur, we can use a condition-controlled loop. The syntax is essentially: while (condition = TRUE) – run some code.

We can replicate a FOR loop we used earlier with a condition-controlled loop.

**Visual Basic .NET**

|  |  |  |
| --- | --- | --- |
| dim countTo = 0  dim startValue = 0  Console.Write("What number do you want to count to?: ")  countTo = cint(Console.ReadLine())  while (startValue <= countTo)  Console.WriteLine(startValue)  startValue = startValue + 1  end while | As you can see, we need some extra things to do the same as a FOR loop.   1. We need to declare a counter variable (start value in this case) 2. We need to increase the ‘counter variable’ manually. | |
| Screenshot | However, as you can see – the results look identical. The question is: What is more efficient? | |
| dim myName = ""  const storedName = "Ceredig"  while (myName <> storedName)  Console.Write("Guess is my name? ")  myName = Console.ReadLine()  end while  Console.WriteLine("This is my name!") | | Screen shot  Here we ask the user to enter data until it is correct/matches other data, and then allow the program to continue. |

### DO UNTIL loops

DO UNTIL loops are fairly similar to WHILE loops, in so far as they execute code until a certain condition is met. However, a DO UNTIL loop **ALWAYS** runs the first iteration of the loop before checking the condition.

|  |  |  |
| --- | --- | --- |
| dim quit = True  do  Console.Write("Please enter a menu choice: ")  loop until (quit = True) | | Here we see that the ‘quit’ Boolean data type (highlighted yellow) is **the same** in each case at the start.  Our check condition for the DO UNTIL loop is whether the variable ‘quit’ is equal to true. If quit is false, then the loop will carry on.  Because quit is set to **true**, quit = True would equate to **true** and therefore the loop would exit.  With a **WHILE** loop, because we check the condition first, the loop would not run…  However, with a DO UNTIL loop, because we check the condition AFTER the ‘do’ part, we would get output from the program. |
| dim quit = True  while (quit = False)  Console.Write("Please enter a menu choice: ")  end while | |
| **DO UNTIL RESULT:** | Screen shot | |
| **WHILE**  **END WHILE RESULT:** | Screen shot | |

**Infinite loops**

It is often easy to create infinite loops without meaning to! These errors will compile and are Run Time errors (i.e. only apparent when the program runs). Most often these are down to logic errors.

|  |  |
| --- | --- |
| dim numberOfParrots = 1  while (numberOfParrots <= 1)  Console.WriteLine("This parrot is dead!")  numberOfParrots = numberOfParrots - 1  end while | |
| Here the user is trying to get the program to print out the number of dead parrots. However, a LOGIC error means that this will loop infinitely! | Screen shot |

# Using exit and continue key words

***The use of exit in loops***

There is a keyword, exit which, when used, will automatically jump out of a code construct that you are using. The use of exit in programming can make the flow of control **very hard to follow** and is considered bad practise by professional coders. It is better to manage the exit from LOOPS, IF statements, etc. through other means (e.g. meeting conditions) rather than the use of exit.

***The use of the continue keyword***

There is a keyword, continue which, when used, will automatically return to the top of the construct, and trigger the next iteration – e.g. any counters will update, and a new iteration starts. Again, this may lead to confusion in the flow of control between constructs. However, it can lead to efficiencies in programming, by avoiding execution of code.

### Example of exit and continue in a for loop

|  |  |
| --- | --- |
| dim countTo = 13  dim countStop = 9  for counter = 0 to countTo  if (counter MOD 2 = 0) then  continue for  elseif (counter = countStop) then  exit for  else  Console.WriteLine("This parrot is number " & counter)  end if  next counter | |
| Here we use the two key words for two purposes.  continue foris used to return to the start of the FOR loop if a number is even. In this case, if continuewas not used, the program would check the rest of the IF statement before looping and print out "This parrot is number X".  exit foris used to exit the FOR loop once we reach 9 (and parrot 9 will not be printed). | Screen shot  **NB:** We could **avoid** using exit here by replacing it with: countTo = 9 |

# The use of basic string manipulation

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| stringname.length  subject = "Computer Science"  subject.length | This gets the length of a string.  subject.length will return 15 |
| stringname.substring(startingPosition, numberOfCharacters)  subject.substring(3,5)  subject.left(4)  subject.right(3) | This gets a substring, but the string will start at the 0th character.  subject.substring(3,5) will return "puter"  subject.left(4) will return "Comp"  subject.right(3) will return "nce" |
| stringname.upper  stringname.lower  subject.upper  subject.lower | This converts the case of the string to either upper or lower case.  subject.upper will return "COMPUTER SCIENCE"  subject.lower will return "computer science"  . |
| ASC(character)  CHR(asciinumber)  ASC(A)  CHR(97) | This converts to and from ASCII.  ASC(A) will return 65 (numerical)  CHR(97) will return "a" (char)  Uppercase letters and lowercase letters have different ASCII values as does numbers represented in a string. |
| someText="Computer Science"  print(someText.length)  print(someText.substring(3,3))  16  put | Here length of the variable is printed along with the 3 characters 3 character in for 3 characters. |

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim messageA = "This part of the "  dim messageB = "message is broken in "  dim messageC = "to three separate strings!" | Here we have a message stored as 3 separate strings. |
| dim messageA = "This part of the "  Console.WriteLine(messageA.Length) | We use String.Length to count the individual characters (including spaces!) within the string.  Screen shot |
| dim messageA = "This part of the "  Console.WriteLine(messageA)  Console.WriteLine(messageA.ToUpper()) | We can use the .ToUppper method to change the case of a string.  Screen shot |

### Placeholders

|  |  |
| --- | --- |
| You can use references for values within strings, to make formatting easier: | |
| dim duckWeight = 34  dim suspectWeight = 35  Console.WriteLine("The duck weighs " & duckWeight & ". However the suspect weighs " & suspectWeight & "!")  Console.WriteLine("The duck weighs {0}. However the suspect weighs {1}!", duckWeight, suspectWeight) | |
| Screen shot | You can see that both lines of code produce the same output, but arguably, using the second method is easier to do, and clearer to follow. |

### ‘Contains’ within a string

|  |  |
| --- | --- |
| dim messageA = "This part of the "  dim result as boolean  result = messageA.Contains("This")  Console.WriteLine(result)  result = messageA.Contains("this")  Console.WriteLine(result) | You can also check to see if strings contain certain values/strings.  This example checks for the word "This" in the string "This part of the " variable.  Note than comparisons are case sensitive.  Screen shot |

### ASCII values

|  |  |
| --- | --- |
| dim messageA = "This part of the "  for index = 0 to MessageA.length - 1  dim asciiValue = ASC(MessageA.substring(index, 1))  Console.WriteLine(asciiValue)  next index  dim messageA = "This part of the "  for each c as string in messageA  dim asciiValue = ASC(c)  Console.WriteLine(asciiValue)  next c | You can return ASCII values of characters.  Screen shot |

|  |  |
| --- | --- |
| Console.Write("Enter your value: ")  dim value = integer.Parse(Console.ReadLine())  dim character = CHR(value)  Console.WriteLine(character) | To find the character an integer refers to, you can use the following.  This gives the result:  Screen shot |

# Substrings

### String cutting from the left

|  |  |
| --- | --- |
| dim messageA = "This part of the "  dim subMessage = left(messageA, 6)  Console.WriteLine(subMessage) | This cuts from the left, starting at index 0, and then taking 6 characters, and saving it as the new string.  Screen shot |

### String cutting from the right

|  |  |
| --- | --- |
| Console.Write("Please enter a string: ")  dim messageA = Console.ReadLine()  dim subMessage = right(messageA,6)  Console.WriteLine(subMessage) | Cutting from the right requires us to know how long the string is. If you know this already, this is easier. If not, then you need to work out the string length first using the String.Length method.  Then you can count back the number of characters you want to cut! Screen shot |

### File handling

File handling requires two classes to be used. StreamReader and StreamWriter. Each class needs to be instantiated during execution.

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| myFile = open("sample.txt")  x = myFile**.**readLine()  myFile.close() | To open a file to read from open is used and readLine to return a line of text from the file. |

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim myFile = new StreamReader("MyTextFile.txt")  dim textFileLine = myFile.ReadLine()  Console.WriteLine(textFileLine)  myFile.Close() | |
| The above code creates the link to read from the file using the StreamReader. Once this is created, we can then read from the file.  This example reads the first line from the file and then save it as the variable ‘textFileLine’. It then prints the variable to screen, as show on the right. | Screen shot |

# The use of close

If we do not close the file after a StreamReader or StreamWriter then the link to the file stays open, and the computer ‘remembers’ where it was up to with that file.

|  |
| --- |
| dim myFile = new StreamReader("MyTextFile.txt")  dim textFileLine = myFile.ReadLine()  Console.WriteLine(textFileLine)  ' Here is some other code we do in the mean time.  Console.WriteLine("My other code has just run and now I want the first line of the file again!")  textFileLine = myFile.ReadLine()  Console.WriteLine(textFileLine) |
| Here we read the first line, and the code goes off and does something else. If we then came back and wanted to read the first line of the file again later, we would find that it actually now reads the SECOND line of the file.  Screen shot |

To tell Visual Basic .NET that we have finished using the StreamReader, we use the **‘.close()**’ method, which **terminates** the StreamReader that is currently in use. This means that we cannot use it again and need to create a new one.

**It is better to try and read all of the file that you will need in one "operation" and then close the file. You can save the file into an array or a string for later, and use that variable within the program. This is much more effective!**

### Reading a file line by line

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| newFile("sample.txt")  myFile = open("sample.txt")  while NOT myFile.endOfFile()  print(myFile.readLine())  endwhile  myFile.close() | newFile() is used to create a new text file with the name given in the brackets. Here we create a new file called sample.txt  readLine is used to return a line of text from the file.  endOfFile()is used to determine the end of the file. The example will print out the contents of sample.txt |

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim myFile = new StreamReader("MyTextFile.txt")  while NOT myFile.endOfStream  dim textFileLine = myFile.ReadLine()  Console.WriteLine(textFileLine)  end while  myFile.Close() | |
| Here we set up two variables. One holds the value that will be read from the file. The other is a Boolean to flag when file is empty.  We use a WHILE loop here, as we do not always know how many lines there may be in the file. If we do – we can use a FOR loop.  We then read a line from the file and use the IF statement to check what happens. If the line is NOT NULL (i.e. contains text) it will print that line out. Otherwise, it changes the flag to ‘true’, indicating the file is empty and then the WHILE loop will not iterate again. | Screen shot  **Note: if you have a file that has empty lines in between text, this method will not work for you!** |

### Reading a whole file at once

|  |  |
| --- | --- |
| dim textFileAll = File.ReadAllText("MyTextFile.txt")  Console.WriteLine(textFileAll) | |
| This reads ALL of the file content into a single variable.  Note that here it does not matter if there are line breaks in between lines of text. | Screen shot |

### Write to a file (overwrite)

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| myFile = open("sample.txt")  myFile.writeLine("Hello World")  myFile.close() | To open a file to write to, open is used and writeLine to add a line of text to the file. In the example, Hello world is made the contents of sample.txt (any previous contents are overwritten). |

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim myFile = new StreamWriter("MyTextFile.txt")  myFile.WriteLine("My new line of text!")  myFile.Close() | |
| By default, Visual Basic .NET will OVERWRITE to a file, if you use the above code.  The file I have used is the SAME file that had the Monty Python quotes in. Note now, that the content has been **overwritten**. | Screen shot |

### Write to a file (append)

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim myFile = new StreamWriter("MyTextFile.txt", True)  myFile.WriteLine("My new line of text!")  myFile.Close() | |
| The use of the keyword true as an option within the StreamWriter instantiation allows us to now ADD extra lines to a file that already exists.  As we can see on the right, we already have text within the text file, and we have now added "My new line of text!" to the end of the file. | Screen shot |

# The use of arrays

Arrays can be single, or multi-dimensional. However, in Visual Basic .NET arrays are static, in so far as once you have said that an array is 5 elements in size, you cannot add a 6th element.

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| array names[5]  names[0]="Ahmad"  names[1]="Ben"  names[2]="Catherine"  names[3]="Dana"  names[4]="Elijah"  print(names[3])  array board[8,8]  board[0,0]="rook" | Arrays will be 0 based and declared with the keyword *array*.  Example of a 2D array: |

### Declaring an array

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim peopleAges(0 to 4) as integer  dim myMapGrid(0 to 4, 0 to 4) as string | Here we have created an empty 1D and a 2D array. The syntax is not quite the same as creating a basic data type variable. |

### Initialising an array with data items

|  |
| --- |
| dim peopleAges() as integer = {5,4,3,2,1}  dim peopleAges2() = {5,4,3,2,1}  dim myMapGrid(,) as string = {{"A1","A2","A3"},{"B1","B2","B3"},{"C1","C2","C3"}}  dim myMapGrid2(,) = {{"A1","A2","A3"},{"B1","B2","B3"},{"C1","C2","C3"}} |
| Above shows how we can both initialise a variable to be of type array and **also** add data items in when initialised. There are two ways of doing this – one slightly longer than the other. Both work! The first way is perhaps viewed as more "formal" as it clearly shows the type of array being declared again in the assignment. |

### Assigning data items to an array after initialisation

|  |  |
| --- | --- |
| dim peopleAges(0 to 4) as integer  peopleAges(0) = 31  peopleAges(1) = 24  peopleAges(2) = 12  peopleAges(3) = 68  peopleAges(4) = 44 | After we have initialised the array, we can add data items, of that data type, into the array.  **All of the data items within an array must be of the same type.**  Here we have added 5 ages to our array.  *Remember that an array index starts at [0]!* |
| dim namesAndAges(0 to 2, 0 to 1) as string  namesAndAges(0, 0) = "Jill"  namesAndAges(0, 1) = "23"  namesAndAges(1, 0) = "Rashpal"  namesAndAges(1, 1) = "43"  namesAndAges(2, 0) = "Leoung"  namesAndAges(2, 1) = "55" | With a 2D array, we need to make sure that:   1. All the data items are of the same type – therefore ages here are stored as **strings** 2. We remember the dimensions of the array when assigning data! |
| **Trying to assign a variable outside of the array boundary**  dim namesAndAges(0 to 2, 0 to 1) as string  namesAndAges(0, 0) = "Jill"  namesAndAges(0, 3) = "23" | The code on the left shows us that we have tried to assign ‘23’ to [0,3] in the array.  The array’s bounds are [3,2] and therefore [0,3] does not exist. It generates the following **run time** error shown below.  **These errors do not show in the IDE**, and will simply result in the program crashing once executed as shown below. |
| **Array out of bounds run time error:**  Screen shot | |

### Sorting arrays

|  |  |
| --- | --- |
| Sorting arrays is usually useful and you can do this by using the Array.Sort() method. However, once sorted, the array will remain in a sorted state | |
| dim peopleAges() = {34,65,12,43,65,76,85,34,11}  Console.WriteLine("Unsorted array items:")  for each age in peopleAges  Console.WriteLine(age)  next age  'Sort the array using Array.Sort()  Array.Sort(peopleAges)  Console.WriteLine("Sorted array items")    for each age in peopleAges  Console.WriteLine(age)  next age | Screen shot |
| You can also use the sort method on Strings as shown below | |
| dim peopleNames() = {"Eric", "Jessica", "Abdul", "Chia", "Chen", "Douglas"}  'Sort the array using Array.Sort()  Array.Sort(peopleNames)  Console.WriteLine("Sorted array items")  for each name in peopleNames  Console.WriteLine(name)  next name | Screen shot |

### Reverse sorting arrays

|  |  |
| --- | --- |
| Sorting arrays in reverse order can be achieved by a two step process.   1. Sort in order 2. Reverse the array | |
| dim peopleAges() = {34,65,12,43,65,76,85,34,11}  'Sort the array using Array.Sort()  Array.Sort(peopleAges)  'Reverse the order using Array.Reverse()  Array.Reverse(peopleAges)  Console.WriteLine("Reverse sorted array items")  for each age in peopleAges  Console.WriteLine(age)  next age | Screen shot |
| **The same works for strings in reverse.** | |
| dim peopleNames() = {"Eric","Jessica","Abdul","Chia" ,"Chen","Douglas"}  'Sort the array using Array.Sort()  Array.Sort(peopleNames)  'Reverse the order using Array.Reverse()  Array.Reverse(peopleNames)  Console.WriteLine("Reverse sorted array items");  for each name in peopleNames  Console.WriteLine(name)  next name | Screen shot |

### Printing arrays

You can print from arrays in two different ways.

1. Using a FOR loop
   1. Used if you already know before code execution, how long each array is
   2. You want to cycle through an array in a specific order (e.g. a 2D array, or in reverse)
2. Using a FOR EACH loop
   1. Easier syntax
   2. Will only print from "top to bottom"
   3. Will do 2D arrays, but only in a specific order

### Using a FOR loop

|  |
| --- |
| **1-dimensional Array**  dim mySweetList = {"Refreshers", "Smarties", "Ferrero Rocher", "Starburst" }  for index = 0 to 3  Console.WriteLine("You own: {0}", mySweetList(index))  next index |
| Screen shot |
| **2-dimensional Array**  dim mySweetList(,) = {{ "Refreshers","Smarties","Ferrero Rocher","Starburst" },  { "Gold bar","Rocky Road","Fruit Pastilles","Moam" }}  for index2 = 0 to 3  for index1 = 0 to 1  Console.WriteLine("You own: {0}", mySweetList(index1,index2))  next index1  next index2 |
| Screen shot |

### Using a FOR EACH loop

|  |
| --- |
| **1-dimensional Array**  dim mySweetList() = {"Refreshers", "Smarties", "Ferrero Rocher", "Starburst" }  for each sweet in mySweetList  Console.WriteLine("You own: {0}", sweet)  next sweet |
| Screen shot |
| **2-dimensional Array**  dim mySweetList = {{ "Refreshers", "Smarties", "Ferrero Rocher", "Starburst" },  { "Gold bar", "Rocky Road", "Fruit Pastilles", "Moam" }}  for each sweet in mySweetList  Console.WriteLine("You own: {0}", sweet)  next sweet |
| Screen shot |

# The use of records to store data

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| array myClass[5,4]  myClass [0,0] = "Name"  myClass [0,1] = "Class"  myClass [0,2] = "Age"  myClass [0,3] = "FavColour" | Records can be written as 2D array.  The first ‘row’ would contain the field names. |
| myClass [1,0] = "Sunita"  myClass [1,1] = "3B"  myClass [1,2] = "15"  myClass [1,3] = "Green" | We can then use each ‘row’ underneath to contain a record. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Class** | **Age** | **FavColour** |
| Sunita | 3B | 15 | Green |
| Gerry | 3B | 15 | Blue |
| Rashford | 3A | 16 | Yellow |
| Bill | 3D | 14 | Mauve |

**Visual Basic .NET**

We already know how to set up a 2D array, and to add data to an array location.

|  |  |
| --- | --- |
| dim myClassArray(0 to 4, 0 to 3) as string  for row = 0 to 4  for col = 0 to 3  Console.WriteLine("Enter value for [{0},{1}]: ", row, col)  myClassArray(row, col) = Console.ReadLine()  next col  next row | |
| As you can see, the FOR loop navigates the array and allows data entry for each data location within the array. | Screen shot |

### Printing a record structure

We can use a FOR loop to print these out in to a format that "looks like" a table through use of inserting TAB characters within the output statements.

|  |  |
| --- | --- |
| dim myClassArray = {{"Name", "Class", "Age", "fColour"},  {"Sunita", "3B", "15", "Green"},  {"Gerry", "3B", "15", "Blue"},  {"Gordon", "3A", "16", "Yellow"},  {"Bill", "3D", "14", "Mauve"}}  'Use a nested FOR loop to cycle through the array  for row = 0 to 4  for col = 0 to 3  'Print out each data item in the array using an alignment value  'the -10 makes the column 10 characters wide, left justified.  Console.Write("{0,-10}", myClassArray(row,col))  next col  'We had to use the Console.Write() to avoid adding in 'returns' after each  'line, therefore we need to add a line break in the outer loop.  Console.WriteLine()  next row | |
| As you can see, this gives us a nicely formatted table-styled print out. However, this does depend on using data items that are **under** the specified alignment value. | Screen shot |

# How to use sub programs

Sub Programs can be used to separate commonly used code into a separate code blocks. This can help us create easy to follow and efficient programs.

There are **two types of sub programs.**

1. Sub: Does not return a value *In the OCR Exam Reference guide this is named procedure*
2. Function: Does return a value

We can also make sub programs more complex by passing variables into them. When we define a sub program, we show the **parameters** that will be used. When we run the sub program, the variables or literals passed into the function are called **arguments**.

sub Main(args as string())

'This is the main body of the program

end sub

}

'This procedure is created outside of the Main program

sub printHello()

Console.WriteLine("Hello!")

end sub

'This function is created outside of the Main program

function returnHello() as string

return "Hello!"

end function

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| function triple(number)  cubedNumber = number\*3  return cubedNumber  endfunction  y=triple(7)  procedure greeting(name)  print("hello" + name)  endprocedure  greeting("Gemma") | Here we define a function with a name that takes an argument (number). The calculation is then performed and the function is ended.  Here we can see the argument for the procedure called from main program to print a string including the argument. |

**Visual Basic .NET**

### Creating a procedure

Procedures (sub) **do not return a value.**

|  |  |
| --- | --- |
| sub printHello()  Console.WriteLine("Hello!")  end sub | Here we have defined a sub program that does not return a value by using  sub / end sub  The name of the procedure is highlighted in green – here it is printHello .  The two brackets ‘()’ show that no parameters are needed to run this function. |

### Creating a function

Functions need to state what **variable type** they return.

|  |  |
| --- | --- |
| function myAge() as integer  return 35  end function  function myMessage() as string  return "She turned me into a newt!"  end function | Here we have defined a function that does return a value.  The first function returns an integer, the second returns a string. |

# Sub programs with parameters

### Single parameter functions

|  |  |
| --- | --- |
| function calcVAT (price as decimal) as decimal  return price \* 1.2  end function | Notice now we have included the data type and identifier that we are passing to the function.  Here we use a decimal to use in the function for calculating a VAT total. |

### Multiple parameter functions

|  |
| --- |
| function calcVAT(price as decimal, vatRate as decimal) as decimal  return price \* vatRate  end function |
| Here we declare that the function needs has two parameters. Both of these parameters are decimal. We then return the calculations as a decimal. |

# Using sub programs within a program

Sub programs are great for sectioning out code, as discussed. We have seen that some sub programs may return values. However, unless we **use** the values a function returns, we lose that data.

### Using returned values

|  |  |
| --- | --- |
| 'Get the user number  Console.WriteLine("What number shall I square? ")  dim number = Cint(Console.ReadLine())  'Use the function to double the number  squareNumber(number)  'Print the answer  Console.WriteLine(number)  'Function to multiply a number by itself  function squareNumber (number as integer) as integer  return number\*number  end function | Here we have created a function to square a number (number \* number).  However there is an issue with the programs output as it stands, which is shown here:  Screen shot  The issue lies with the red highlighting. We send the argument to the function – which is the number the user typed in. The function correctly works out number\*number. However, **we do not store the value that the function returns.** |
| **Original**  'Use the function to double the number  squareNumber(number)  **New**  'Use the function to double the number  number = squareNumber(number) | With the modified code, we can now assign the returned value to a variable (here we just overwrite the value of number).  It is important to remember to do this, otherwise you may lose data, or store incorrect data etc. |

|  |  |
| --- | --- |
| 'Get the user number  Console.WriteLine("What number shall I square? ")  dim number = Cint(Console.ReadLine())  'Use the function to double the number  number = squareNumber(number)  'Print the answer  Console.WriteLine(number)  'Function to multiply a number by itself  function squareNumber (number as integer) as integer  return number\*number  end function | |
| Screen shot | This is the result of the updated program code, and as you can see, it now correctly squares the number, stores it as ‘number’ and then prints it to the screen. |

### Where should I use functions?

Functions are for repeated code blocks that may be used many times. Common examples could be:

* Calculations
* Validation
* Conversions
* Recursion

# Casting

As everything from the keyboard in Visual Basic .NET is viewed as a string by the compiler, we need to cast our input into the correct data type. Sometimes Visual Basic .NET will do this automatically, but sometimes it can be hard to predict what it is going to do, so it is better to convert data types explicitly using casting functions.

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| str(3) returns "3"  int("3") returns 3  float("3.14") returns 3.14  real("3.14") returns 3.14  bool("True") return True | Variables can be typecast using the int str, real and bool float functions. |

**Visual Basic .NET**

|  |  |
| --- | --- |
| 'Variables  dim userEntry = "54"  dim castNumber as integer  dim decNumber as decimal  dim backToString as string  'Various casting options  castNumber = cint(userEntry)  decNumber = cdec(userEntry)  backToString = cstr(decNumber)  'Printing the output  Console.WriteLine(userEntry)  Console.WriteLine(castNumber)  Console.WriteLine(decNumber)  Console.WriteLine(backToString) | Here we have option to cast from strings to numerical data types and from numerical data types back to a string.  The output is shown below:  Screen shot  As we can see it makes no different to the printing to screen – but it **does** allow us to now carry out calculations on the numeric data types such as multiplication and division. |

# Random numbers

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| number = random(1,6) | Creates a random number between 1 and 6 inclusive. |
| number = random(-1.0, 10.0) | Creates a random real number between -1.0 and 10 inclusive |

**Visual Basic .NET**

|  |  |
| --- | --- |
| dim randomGenerator = new Random()  'The range is from 0 (inclusive) to 7 (not inclusive)  dim myRandomNumber = randomGenerator.Next(0, 7)  Console.WriteLine(myRandomNumber) | As the random generator is a method within a pre-defined class, we have to create an instance of this class first.  Next we can assign a random number to a variable, using the name of the instance we have created (here it is randomGenerator), the .Next() call and then a range (shown in the brackets.)  The last line prints this variable to screen. |

We can use the same random generator for any random number, and simply change the range of the arguments we pass through to the Random method we use.

|  |  |
| --- | --- |
| dim myRandomNumber as integer  dim randomGenerator as new Random()  myRandomNumber = randomGenerator.Next(-40, 1)  Console.WriteLine(myRandomNumber)  myRandomNumber = randomGenerator.Next(0, 51)  Console.WriteLine(myRandomNumber)  myRandomNumber = randomGenerator.Next(345, 555)  Console.WriteLine(myRandomNumber) | Here we generate 3 random numbers using the same instantiation randomGenerator.  The three numbers we generate are between:   * -40 to 0 * 0 to 50 * 345 to 554   Screen shot |

# Combinations of techniques

Following is an example that uses the majority of techniques within the syntax guide. It combines a DO-WHILE, SWTICH, Random, bool, string, int, functions without parameters, functions with parameters and IF-ELSE statements.

|  |
| --- |
| Public Sub Main()  dim witchWeight = 0, duckWeight = 0  dim judgement = false  discussion()  do  Console.WriteLine("Witch assessment menu: ")  Console.WriteLine("1 - weigh a witch ")  Console.WriteLine("2 - weigh a duck ")  Console.WriteLine("3 - pass judgement on the witch")  Console.Write("Please enter a menu choice: ")  dim menuchoice = Console.ReadLine()  select case menuchoice  case "1"  witchWeight = generateWeight()  Console.WriteLine("Your witch weighs: {0}", witchWeight)  case "2"  duckWeight = generateWeight()  Console.WriteLine("Your duck weighs: {0}", duckWeight)  case "3"  if (witchWeight = 0) then  Console.WriteLine("It appears that you haven't weighed your witch yet!")  elseif (duckWeight = 0) then  Console.WriteLine("It appears that you haven't weighed your duck yet!")  else  judgement = weightCheck(witchWeight, duckWeight)  end if  end select  loop until judgement  end sub    sub discussion()  Console.WriteLine("So, logically, If she weighs the same as a duck...")  Console.WriteLine("She's made of wood!")  Console.WriteLine("And therefore?")  Console.WriteLine("A witch!")  end sub  function weightCheck(witch as integer, duck as integer) as boolean  Console.WriteLine("We have compared the witch and the duck!")  if (witch >= duck) then  Console.WriteLine("A witch! Burn the witch!")  return true  else  Console.WriteLine("Not a witch... try another one!")  return false  end if  end function  function generateWeight() as integer  dim generator as new Random()  dim newWeight = generator.Next(1, 100)  return newWeight  end function |
|  |
| Screen shotHere is a sample of some of the output from the program.  As you can see, we now have a great program the weighs a suspected witch, weighs a duck and then compares which is heavier.  If the duck is heavier, then the witch is condemned.  If the duck is lighter, the witch survives!  If you do not weigh either the witch or the duck, it does not allow you to progress to ‘passing judgement’ and tells you as such. |

# Recursion

Recursion is when a function calls itself during its execution:

|  |  |
| --- | --- |
| Public Sub Main()  countdown(10)  end sub  sub countdown(tMinus as integer)  if (tMinus ==0)  Console.WriteLine("\*\*\* Blast Off!! \*\*\*")  else  Console.WriteLine(tMinus)  countdown(tMinus-1)  end if  end sub | |
| Screen shot | The flow of control starts with the Main method. We have one line of code – countdown(10). This calls the function countdown, using the argument of 10 (i.e. we want the countdown to start at 10).  The function ‘countdown’ will print a message if the variable is equal to 0 (in this case ‘blast off’).  However, if it isn’t equal to 0 then it prints the variable tMinus, and then call itself using the variable-1.  Effectively this means that you would have 10 of the functions running, all ‘nested’ inside each other. |

# Scope

Variables have scope. Scope is the ‘visibility’ of a variable to other parts of the program. Scope is a very powerful thing, and can be used effectively to manage memory and stop mistaken access to variables.

### Local variables

**Example 1:**

|  |  |
| --- | --- |
| Public Sub Main()  Console.Write("Enter your name: ")  dim name = Console.ReadLine()  printName()  end sub  sub printName()  Console.WriteLine("Your name is {0}", name)  end sub | The code on the left will not compile.  We have a variable, ‘name’ as a string, which allows the user to enter a string into the program.  We want to use this variable later on in a procedure to print out a message. However, the **scope** of the variable is **local** to that procedure. Therefore, only code that uses the ‘name’ variable within the Main program will be able to ‘see’ the variable.  The bottom yellow line will not work, as name cannot be ‘seen’ by this method. |

**Example 2:**

|  |  |
| --- | --- |
| Public Sub Main()  getName()  Console.WriteLine("Your name is " & myName);  end sub  sub getName()  dim myName = Console.ReadLine()  end sub | The code on the left will also not compile.  The variable ‘myName’ is created within the procedure ‘getName’. However, it can firstly **only** be seen within this procedure.  Secondly, after the procedure runs, the variable is **destroyed** and ceases to exist. Therefore it cannot be used within the Main program. |

### Avoiding scope issues

The best way to avoid scope issues is to declare any variables you will need throughout your program within the main program. You can then use them within sub programs you create as **arguments** that you pass into the sub programs (see the section on sub programs). However, you may then also need to **return** the results of the function to overwrite these variables with a new value.

The advantage of doing it this way is that you have to physically pass and return variables; this means you can keep a close watch on what is going on!

Another way could be to use global variables.

**Global variables**

Global variables in Visual Basic .Net are more complex to set up. They can also be viewed as "slack" lack of programming. If you set every variable as global – you can get access to them anywhere within the program. Whilst this may seem a good idea, you can potentially end up overwriting variables by mistake.

|  |  |
| --- | --- |
| public module Module1  const vatRate as decimal = 1.2  dim itemCost as decimal  public sub Main()  Console.Write("What is the cost of the item without VAT: ")  itemCost = cdec(Console.ReadLine())  Console.WriteLine("The true cost of your item is: {0:f2}", vatRate\*itemCost)  end sub  end module | |
| In the code above, we have created one global constant and one global variable (highlighted in green) and then used them within the Main program. They could also be used in any other procedure or function in the module. If you declare a local variable of the same name, this will take precedence for that sub program. | Screen shot |

# The use of SQL to search for data

**OCR Exam Reference Language**

|  |  |
| --- | --- |
| SELECT  FROM  WHERE | SELECT LastName  FROM Customers  WHERE LastName = "Smith"; |

**SQL**

This example assumes there is a database created called "Customers" with columns called:

* CustomerID
* CustomerName
* ContactName
* Address
* City
* Country

|  |  |
| --- | --- |
| SELECT CustomerID FROM Customers | This selects the CustomerID field from the Customers database. |
| SELECT *ContactName*,*Address* FROM *Customers* WHERE *ContactName = "Mr Creosote"*; | This selects the ContactName and Address columns from the Customers table and then specifically looks for a Mr Creosote in the ContactName field. |