www.tongatapu.net.to

QBASIC TUTORIAL

18 January 1999

GOALS AND OBJECTIVES

This Qbasic Tutorial provides an introduction to Computer Programming through the use of the Microsoft Qbasic programming language.

At the conclusion of this module, students should be able to create their own Qbasic programs from a list of problem tasks.

MODULE OUTLINE

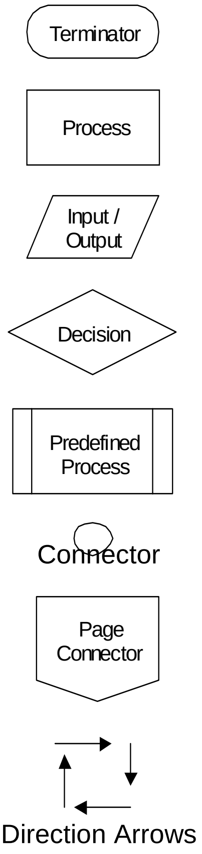
Syntax and Semantics
Variable Storage
Mathematical Expressions
A Problem Resolution Process
Flow Control
Making Comparisons – Conditionals
Repetitions – the FOR loop
Repetitions – the DO loop
Repetitions – the EXIT condition loop

Notes:
These notes are especially developed to assist teachers and students in classroom instruction with exercises to re-enforce instruction.

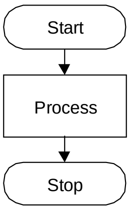
Students with high language skills and familiarity with computers should be able to independently complete this tutorial.

FLOW CHARTING

ANSI Flow Chart Symbols



The notes make extensive use of flow-chart symbols to describe, explain the flow, or sequence of program instructions. Standard symbols are used, so other people can read your flow chart diagram, and you can read other peoples' diagrams.



Program instructions are described through the flow-chart with instructions grouped into different symbols, and the 'flow' or sequence of instructions being used directed through the use of connecting arrows.

Terminator. The terminator symbol marks the beginning, or ending of a sequence of instructions. This is often used when marking the beginning and ending of the program.

Process. Marks instructions that are processed such as calculations and declarations. For our purpose, if you cannot figure out which symbol to use, then use this symbol as a placeholder until you can be more certain which is the better flow-chart symbol to use. We will use it for when we make mathematical calculations and declaring variables.

Input/Output. Marks instructions to perform data input (bring data into the program from outside) or output (send data out from the program). We will use this when we ask the user for keyboard input and when we display information to the screen or printer.

Decision. Marks instructions where the program makes a decision. Decisions are the only symbols allowed to have more than one flow out of the symbol. Decisions should have an outside flow of **yes** and **no**. We will use this symbol when comparing different data items.

Predefined Process. Marks a group of instructions. A predefined process can be used to specify that the specifics of these instructions are already known, or are shown in some other place. We will use this symbol to simplify larger programs, where we already know what is to be done and do not want the flow-diagram to take up too much space.

Connector. Joins different parts of the chart together. This is used when the chart gets big and the number of lines may become confusing. The connector circle is labelled with the label that will identify the ingoing connector.

Page Connector. Joins different pages of a chart. Use the page connector at the bottom of the page, using the number of the page where the flow chart will continue as the label. On the top of the connected page, place a page connector symbol at the top of the flow-chart

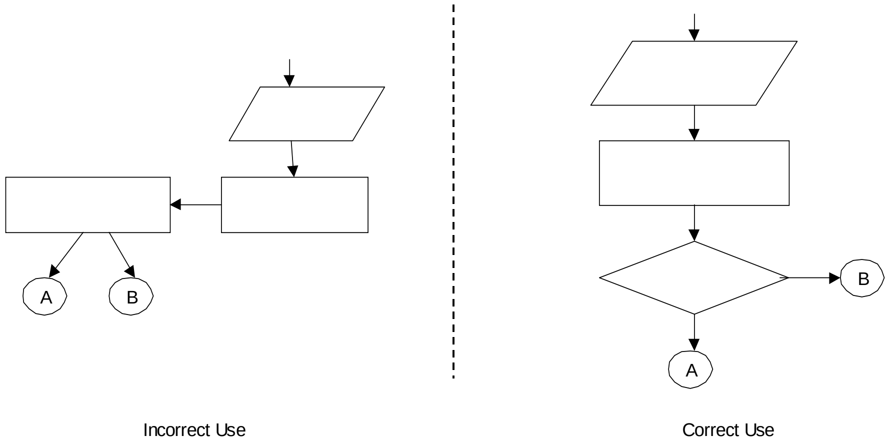
Direction Arrows. These arrows connect the different symbols, identifying in which direction the instructions will be processed.

CASE STUDY

Correct and Incorrect Use of Flow-chart symbols.

Observe in the sample diagram that the following errors are in the 'Incorrect' use column.

- Flow should be from top to bottom, and should not split sideways except through the use of a decision, or diamond, symbol.
- Only a decision can have multiple outgoing connections (arrows).

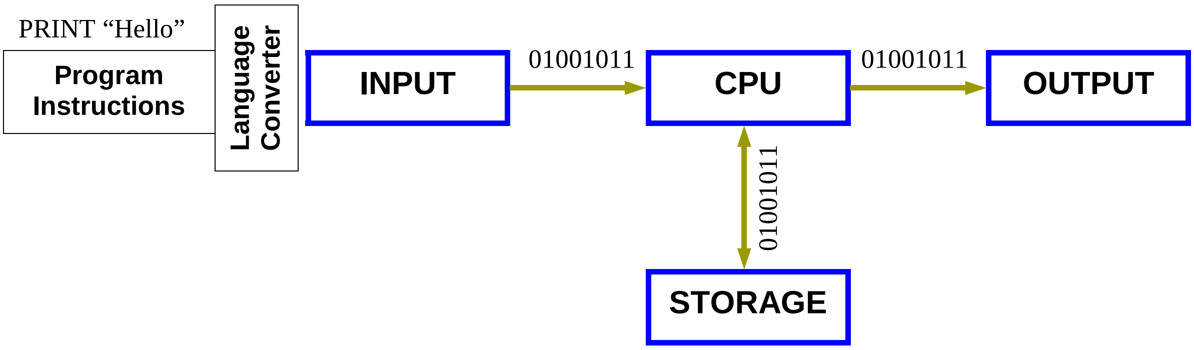


Programming Languages

Computers only comprehend binary, 0's and 1's so writing instructions for a computer to perform requires sending the CPU and peripheral devices a sequence of 0's and 1's in the pre-determined sequence that will cause the computer to perform what tasks is required of it.

0's and 1's mean very little to most human beings, and putting together a correct sequence of 0001101011 01101011011 is problematic because it is very easy to make a mistake. A mistake of putting a 1 where a 0 should be is minor to a human but very important/significant for computers.

Computer Programming Languages were designed to allow humans to work in a language more similar to what we are used to. The human writes the English like commands from the Programming Language and the Language tools convert these instructions into the 0's and 1's that the computer can understand.



LANGUAGE RULES

Computer Programming Languages maintain a number of rules that are common to the regular human language. To be a language it must have rules that prevent ambiguity, misunderstanding, otherwise computers will behave differently from each other.

Grammar. Computer Languages have grammar rules, commonly termed "Syntax" which determines the meaning of the instructions.

Summary Commands in the QBasic Integrated Development Environment (IDE)

Table 7.1 Menu Shortcut Keys

Key	Menu Option or Selection
F1	**H**elp
F2	**S**UBs...
F3	**R**epeat Last Find
F4	**V**iew | **O**utput Screen
F5	**R**un | **C**ontinue
Shift+F5	**R**un | **S**tart
F6	**W**indow (Change between Windows)

The QBasic development environment supports the use of shortcut keys to quickly access menus. Table 7.1 lists the menu shortcut keys. The table uses the bar character "|" to specify the major menu selection and the sub-menu. For example, the F5 function key is the shortcut key for selecting the **R**un Menu and then the **C**ontinue command.

Use of QBasic does not require memorising these shortcut keys, and as you spend more time with QBasic these keys will help you make use your time more effectively.

A program in QBasic is entered into the Edit Window, and the programmer tells QBasic to execute (run) the program instructions by using the command listed at the status bar, bottom of the screen `<F5=Run>` or by selecting the Run menu.

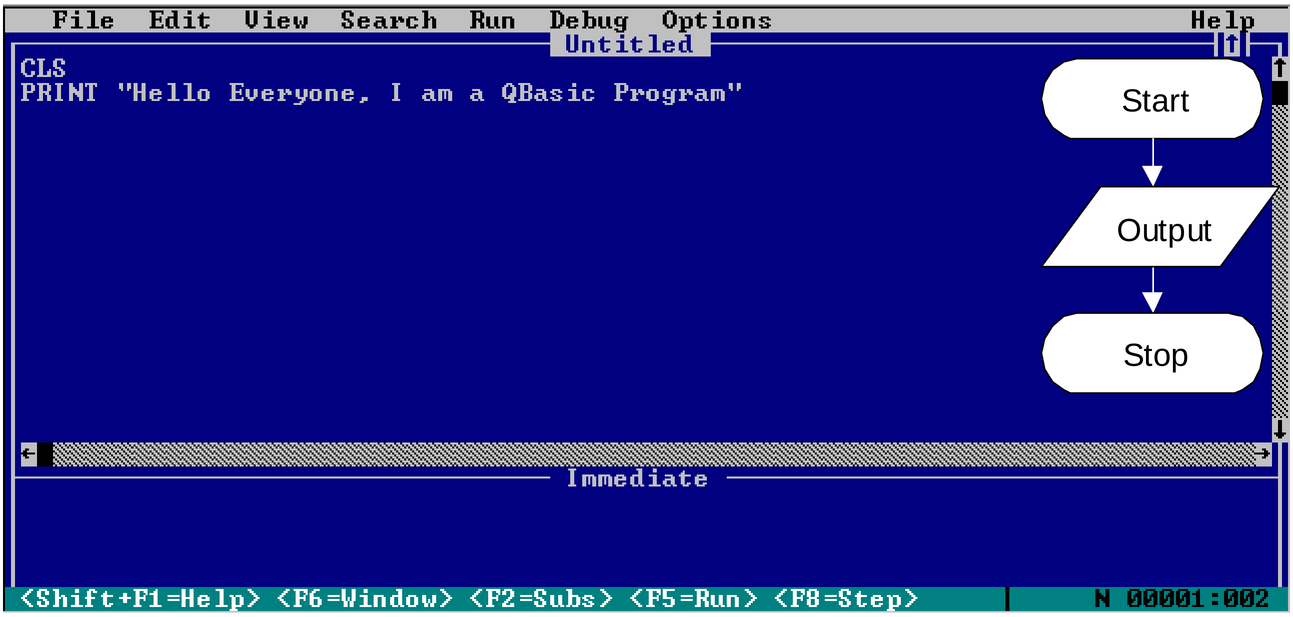
The QBasic Development Environment is set into 3 tiled windows. The top window (usually hidden and only available when you ask for it) is used to display HELP information. The middle window is the Editing window where program instructions are listed, edited. The lower window is the "Immediate" window where QBasic commands/instructions can be entered for immediate execution.

The output screen, where the program puts out information, is hidden behind the Integrated Development Environment (IDE) and can be viewed by using the `<F4=Output Screen>`.

BEGINNING WITH A SIMPLE PROGRAM: PRINT

Listing. Our first program

```
CLS
PRINT "Hello Everyone, I am a QBasic Program"
```


File Edit View Search Run Debug Options Help
Untitled
Start
Output
Stop
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
N 00001:002

```
CLS
PRINT "Hello Everyone, I am a QBasic Program"
```

Screenshot 7.1: The QBasic Integrated Development Environment (IDE)

The above QBasic program when typed in and executed (by Selecting **<F5=RUN>**) will display an output screen as in Screenshot 7.2.

After you "Press any key to continue" you will be back into the Editing window. To look again at the 'output' window, shown in this diagram, use the command **<F4=Output Screen>**

OUR LINE BY LINE REVIEW.

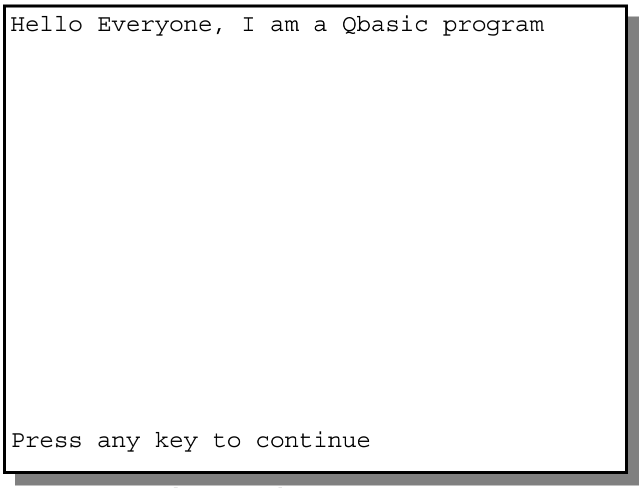
```
CLS
```

is a QBasic instruction to "**CL**ear the **S**creen". This tells QBasic to send to the Screen the instructions required to clear what ever was there previously.

```
PRINT "Hello Everyone, I am a QBasic Program"
```

This is an instruction with additional information. The QBasic instruction is the "PRINT" command with additional information being provided "Hello Everyone, I am a QBasic Program" placed inside quotation marks.

The PRINT instruction is used to print data to the screen or a file. The 'data' to be printed is whatever is set after the PRINT. In this example, "Hello Everyone, I am a QBasic Program" is what PRINT is told to put onto the screen.



```
Hello Everyone, I am a Qbasic program











Press any key to continue
```

Screenshot 7.2: The QBasic Output Screen

SYNTAX CHECKING

Before executing any line of programming, QBasic first checks whether the things you typed in is valid QBasic language. For many languages, this syntax checking is generally processed after all the instructions have been listed (program has been written/coded.)

QBasic has the ability to check each line of programming when the programmer hits the Enter key. This is easily verifiable (confirmed) by typing in a QBasic command in lowercase (eg. cls) and watch QBasic change the command to uppercase (eg. CLS).

INTERPRETED -VS- COMPILED LANGUAGES

The ability of a programming language to check the syntax while the program is being written is usually attributed, or the programming language itself is usually categorised as an "Interpretive Language". Interpretive Languages are so labelled because of their ability to 'interpret' the program instructions while the programmer is developing.

A side-effect, or a result from using an Interpretive Languages, is that the program has to be run from inside the Interpretive Language. In our QBasic example, this means that to run the program we have to be inside QBasic. This requirement by Interpretive Language usually means that programs written in this language are usually slower than if the program did not have to be loaded into the Interpreter.

The counter programming language category is called the "Compiled Language". Instead of checking the syntax as the programmer writes the program, the programmer has to run a separate program (usually called a compiler) which checks the program syntax and converts the text from the program language (compiles) into the binary language of computers. The compiling process generates a program independent of the programming environment which means you can run the program without having to go into the language editor.

COMMON PROGRAMMING ERRORS

When typing in programs, and executing them with F5, a common problem is to incorrectly type in the program code.

Error: `CL S`

`CLS` is very different to the computer from `CL (space) S`

The above is a ‘syntax’ error, or sentence grammar error since QBasic does not know what to do with a `CL` and an `S`. `CL` (space) `S` has no meaning on its own, unlike `CLS` (no spaces).

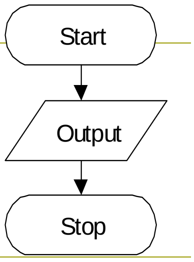
Error: `PRINT Hello Everyone, I am a QBasic Program`

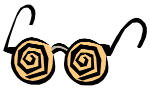
The print command needs the “quotation-marks” to know which things are to be put straight to the screen, and which things to try and find from the computers RAM (memory). In the above example, where there is no quotation marks, the program will try to find memory space that have been labelled as:-
Hello, Everyone, I, am, a, QBasic, and Program.

Make sure you are careful with typing and note taking.

Listing 1.2. Our Second Program – Printing with semi-colons

```
CLS
COLOR 9
PRINT "The Print command starts a new line"
PRINT "Unless you use the semi-colon”
PRINT "Which ";
PRINT "continues on the same line."
```



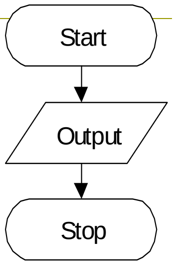


PRINT puts onto the screen the text inside the “quotation marks” and then starts a new line.

There are times when a sentence on the screen looks better if it continues, instead of starting again on the next line. The “;” semi-colon is used in QBasic to tell the PRINT command to continue the next PRINT statement on the same line.

Listing 1.3. Common Programming Errors – Printing with semi-colons

```
COLOR 9
PRINT "1. Where is ;"
PRINT "This Line"

COLOR 2
PRINT "2. Where is ";
PRINT "This Line"
```



COMMON PROGRAMMING ERRORS

When the semi-colon is **inside** the “quotation marks”, it gets printed on the screen like everything else in “quotation marks”.

When the semi-colon is **outside** the “quotation marks”, it tells PRINT to continue the next print instruction on the same line.

```
PRINT USING writes formatted output to the screen or to a file.

PRINT USING formatstring$; expressionlist [{; | ,}]

• formatstring$; A string expression containing one or more
• expressionlist A list of one or more numeric or string expressions to print, separated
  by commas, semicolons, spaces, or tabs.
• {; | ,} Determines where the next output begins:
  ; means print immediately after the last value.
  , means print at the start of the next print zone. Print zones are 14 characters wide.

Example:
    a = 123.4567
    PRINT USING "###.##"; a
    a$ = "ABCDEFG"
    PRINT USING "!"; a$
```

PRETTY PRINTING

QBasic's PRINT USING command gives the programmer more control of how to display information on the screen.

When printing numerical values, QBasic will automatically pick a format to fit as much of the decimal values as it can. PRINT USING gives the programmer specific control of how many decimal values are to be printed out, independent of how many decimals are actually available.

Listing

```
a = 123.4567
PRINT a
PRINT USING "#####.##"; a
PRINT USING "+####.####"; a
PRINT USING "$***"; a
```

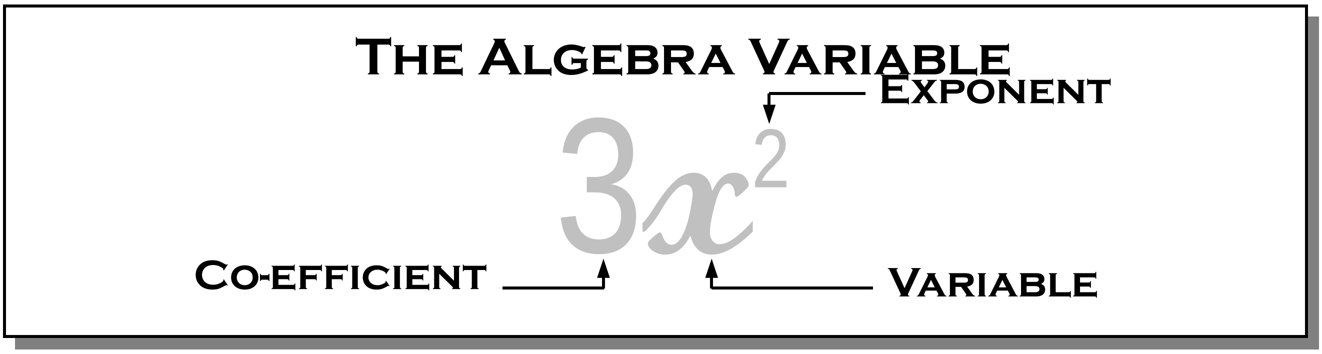
Sample Output

```
123.4567
  123.46
 +123.4567
$%123*
```

#	prints the numeric digit, or a space
$	prints a dollar sign
+	prints a + if the number is a positive number
-	prints a – if the number is a negative number
.	Sets the position of the decimal point. This is useful for lining up the decimal point in a column.

INTRODUCING VARIABLES

Review Existing Knowledge: In Form 3 Mathematics, a ***variable*** is a ***letter*** usually representing an unknown number.



A *co-efficient* is a number multiplying the variable, and
An *exponent* is also called a power or index.

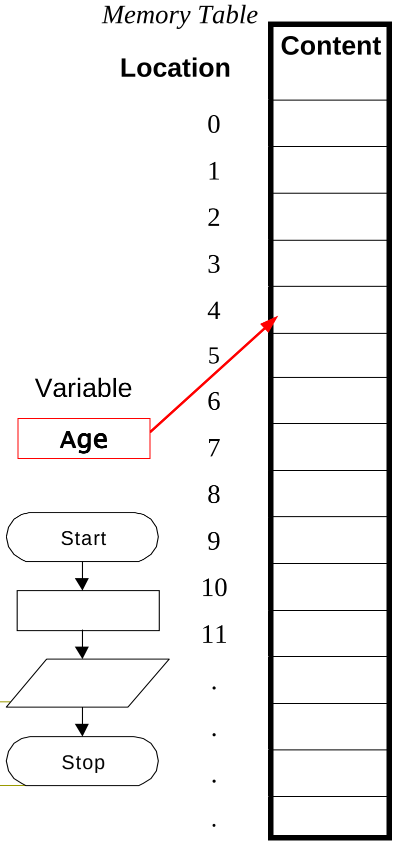
Expanding New Knowledge: Computer programs need space (we will call memory, RAM) to store information the program wants to manipulate or work on.

If we think of computer RAM as a large table of "stuff", a program points to a particular 'cell' or memory address by using a label for the cell, commonly named a '*variable'*.

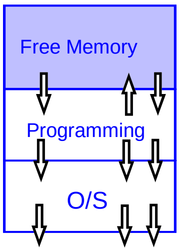
A '*variable*' is a name to refer to a location in the computers RAM, a memory address, or cell. Your program can get the value of the memory address by using the variable name, and can also change the value of the memory address by assigning a new value to the variable.

To create a variable, you just use a name that is **not already used** by the QBasic language. For example, we cannot use PRINT as a variable name because QBasic is already using PRINT to mean something other than a name for a memory address.

Diagram: Computer RAM Memory Table



Location	Content
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
.	
.	
.	
.	



The variable age.

We can create the variable '*age*' in our program by just using it, and we can create the values for age by assigning a value to the variable age.

```
age = 11
PRINT "I am "; age; "years old."
```

BE DESCRIPTIVE

Age - is a good name for a variable, because it is not already used (reserved) by QBasic and because it describes what should be the content of the variable. Whenever we look at the variable 'age' in our program code it looks like a memory address where a number related to the age of a person is stored.

rk11 - will work as a variable name because it is not used (reserved) by QBasic but it means very little to us and can be confusing when we use it.

When you write your programs, think about what the variables will be used for and try to make the variable's name describe what the variable contains. This makes it simpler for you when you need to check your program, and simpler for your friends when you ask them to help you check your program.

Rules on Variable Names

Variables names for QBasic can be whatever fits into the following rules:

- The first character must be a letter from the alphabet
- After the first letter, a variable name can have letters, digits or underscores
- Variable names cannot be words already used by QBasic (reserved) for other purposes

QBasic does not care whether your variable names are upper-case (capital letters), lower-case (small letters), or a mix of the letters. QBasic will change the lettering style to what ever you use as the last lettering style.

EXAMPLES (CAN USE)

```
GarbageCans, GreenTrees, Room_23
```

EXAMPLES (CANNOT USE)

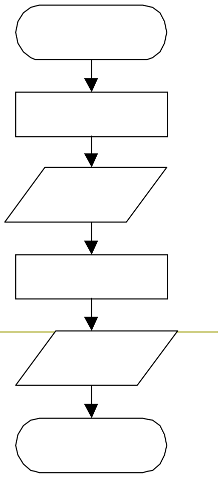
```
23Chicago, PRINT, Room 23, Springsteen~45
```

What is wrong with the above variable names ?

EXERCISES

List FIVE variables that are valid (we are allowed to use)

List FIVE variables that are not valid (we are not allowed to use)



Listing 1.4. Using variables to store data

```
CLS
age = 11
PRINT "I am "; age; "years old."
age = 25
PRINT "But I will grow and soon will be"; age
```

For example, in the above sample program, if you type the last occurrence of the age variable to all upper-case AGE, QBasic will change all "age" variables to upper-case.

Looking through the code execution (desk-check)

```
Line 1.   the screen should clear.
Line 2.   The program creates a variable called 'age' and sets it to point
          to a memory address, cell.  The assignment operator "=" puts the
          value 11 into this memory location.
```

Line 3a. The PRINT statement finds a "I am" is in quotation marks and sends it to the screen.

Line 3b. The PRINT statement finds 'age' by itself, so it checks to see if it is a variable, and gets the value of the memory address to send to the screen.

Line 3c. The PRINT statement finds "years old" in quotation marks and sends it to the screen.

Line 3d. The PRINT statement does not find a semi-colon at the end of the line, so it starts a new line for the next PRINT command.

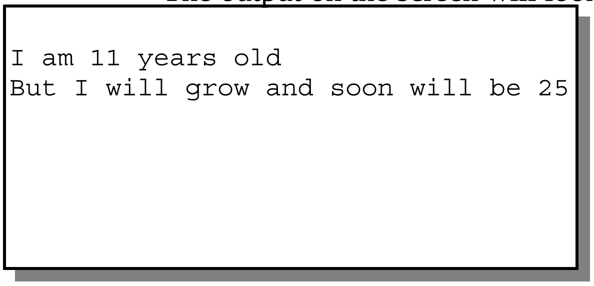
Line 4. The assignment operator sends 25 to the memory address pointed to by 'age'.

Line 5a. The PRINT statement finds "But I will grow and soon will be" in quotation marks so it sends it to the screen.

Line 5b. The PRINT statement finds 'age' by itself, so it checks to see if it is a variable, and gets the value of the memory address to send to the screen.

Line 5c. The PRINT statement does not find a semi-colon at the end of the line, so it starts a new line for the next PRINT command.

The output on the screen will look like the below diagram:



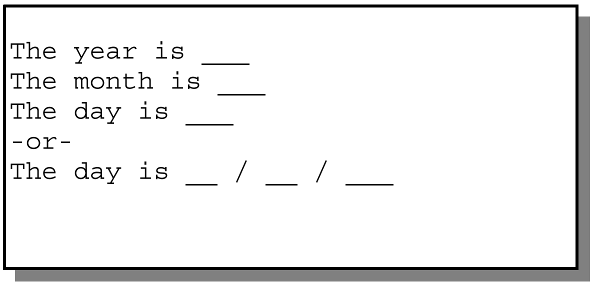
```
I am 11 years old
But I will grow and soon will be 25
```

Mixing the use of variables, and the PRINT semi-colon allows us to combine various bits of information when we display the results to the screen.

Listing 1.5. Printing multiple variables

```
CLS
year = 1921
day = 14
month = 3
PRINT "The year is"; year
PRINT "The month is"; month
PRINT "The day is"; day
PRINT "-or-"
PRINT "The day is ";day;"/";month;"/";year
```

Fill in the details below, how will the variables be allocated in the print display below?



```
The year is ___
The month is ___
The day is ___
-or-
The day is __ / __ / ___
```

STORAGE – ALLOCATING SPACE TO STORE INFORMATION

Storage.
This section assumes completion of other modules on binary numbers and their use in computer storage.

The available computing resources; disk-space, RAM space is limited. The limited resource, and because computers handle text differently from numbers has led to programming languages defining the use and space used by storage locations.

Numbers – are stored as Integer, Long Integer, Single, or Double. Single and Double are variable data-types that can store 'whole numbers' which includes decimal values. Integers do not have decimal values.

String – A string is QBasic's term for storage locations pointed to by variables. Strings can contain numbers, but when they do mathematical formulas do not work on the numbers as they would with numbers stored in variables designed for numbers.

It is a wise programmer who learns to use the appropriate data-type for the purpose they choose their variable, and to select the maximum storage (number of bytes) required.

DIM – DIMENSIONING, EXPLICITLY SPECIFYING THE STORAGE LOCATION.

It is good programming practice, to specify the storage requirements for variables at the beginning of the program.

QBasic uses the keyword DIM to dimension, set aside the memory location to store the data being pointed to by your variable.

data-types
INTEGER, LONG, SINGLE, DOUBLE, STRING, or a user-defined data-type

```
DIM variable AS data-type
```

The DIM keyword is specified at the beginning of the line, followed by the variable name you choose, the keyword AS and the data-type to be used. For example:

```
DIM numb1a AS INTEGER
DIM numb1b AS LONG
numb1a = 1   ' This is a valid use of numb1a
numb1a = 1.1 ' This is not a valid use
```

LONG Integers can hold bigger numbers than the regular integer.

Data Type *(byte size)*	**Minimum Value**	**Maximum Value**
Integer *(2 bytes)*	-32,768	32,767
Long Integer *(4 bytes)*	-2,147,483,648	2,147,483,647

Singles and Double can store numbers which include decimal values. Therefore, it can also hold integers. This number type can hold larger numbers than Integer and Longs

Data Type *(byte size)*	**Minimum Value**	**Maximum Value**
Single-precision *(4 bytes)*		
Positive	2.802597*10^{-45}	3.402823*10^{38}
Negative	-3.402823*10^{38}	-2.802597*10^{-45}
Double-precision *(8 bytes)*		
Positive	4.940656458412465D-324	1.79769313486231D+308
Negative	-1.79769313486231D+308	-4.940656458412465D-324

```
DIM numb2a AS SINGLE
DIM numb2b AS DOUBLE
numb2a = 142
numb2a = 1.0234
```

Strings are set up by using the STRING keyword.

```
DIM brand as STRING
brand = "Nike"
brand = "Estee Lauder"
```

Data Type *(byte size)*	Minimum Value	Maximum Value
String Length *(varies)*	0 characters	32,767 characters

Most computer programming languages require specifying the data-storage requirements before using it. One of BASIC's features, and often criticized, is that it allows the programmer to ***not*** *specify the storage requirements. When BASIC encounters, or comes across a variable that has not been defined (DIMensioned) then BASIC will take a guess at what type of storage it should be. Sometimes it guesses right, other times it makes a mistake.*

SHORTCUTS, EXPLICITLY SPECIFYING THE STORAGE LOCATION.

A short-cut (or short-hand) method for dimensioning storage location is to use QBasic's special symbols after the variable name.

variable%	- The % specifies the variable is an integer
variable&	- The & specifies the variable is a Long
variable!	- The ! specifies the variable is a Single
variable#	- The # specifies the variable is a Double
variable$	- The $ specifies the variable is a String

```
' Using BASIC's short-cut variable dimensioning
' -otherwise known- as suffixes

Numb%   = 4
numbers! = 1027.25
brand$ = "Avon Lady"
```

STRINGS – STORING WORDS AND LETTERS (ALPHANUMERIC DATA)

A simple example for using strings is to store names, such as a person’s name or address. Numbers can be put in a string, but because this storage space is not specified for a number QBasic will handle it different to 'numbers.'

```
name1$ = "Freddy Bloggs"
name2$ = "Freddy Bloggs Sister"
PRINT name1$
PRINT name2$
```

The above program will create two new string variables *name1$* and *name2$* to hold the value given to them above. For example, we are telling the variables to hold *"Freddy Bloggs"*, and *"Freddy Bloggs sister"*. The following PRINT statements will print out the contents of the variables to screen.

Getting Keyboard Input – Letting the User Have a Say

Variables are great, but they really shine (are of extra value) for getting information to and from the user.

We already know how to use the keyboard to enter data and store it within a variable of your choice. We have already used the following statement.

```
CLS

PRINT "Please Enter your Name"
INPUT name$

PRINT "Hello ";name$
```

The above program will simply ask you for your name and when the **[Return]** or **[Enter]** key has been pressed, the variable information is printed to the screen.

INPUT can also print-out information inside quotation marks as '*prompts*' to the user for what we want as input. (just as if we were using the print command.)

We can modify the program to not contain the " ?" question mark.

```
INPUT "",name$
```

By modifying the INPUT command to use the comma instead of INPUT on its own, or using the ";" semi-colon

The question ? disappears.

```
CLS
INPUT "Good-day, what is your name ?", name$
PRINT "Hello"; name$; ", how are you ?"
```

Mixing INPUT and OUTPUT, using the ";" semi-colon is a tool for communicating with the user, formatting the output of our program onto the screen.

```
CLS
INPUT "Enter a year "; year
INPUT "Enter a day "; day
INPUT "Enter a month "; month

PRINT "The year is"; year
PRINT "The month is"; month
PRINT "The day is"; day
PRINT "-or-"
PRINT "The day is ";day;"/";month;"/";year
```

The above program will display something such as the below (using input of 1998, 8, 10)

```
Enter a year 1998
Enter a day 8
Enter a month 10

The year is 1998
The month is 8
The day is 10
-or-
The day is 8 / 10 / 1998
```

The Comma (,) allows the INPUT command to accept multiple variable values in the same entry line.

```
PRINT "Enter your first and last name separated by a comma"
INPUT "", name$, lname$
PRINT "Hello "; lname$; ", "; name$
```

By using a comma when entering data, the above INPUT command separates the information entered into the variable names listed in its list. An example output is listed below:

```
Enter your first and last name separated by a comma
Samiuela, Taufa
Hello Taufa, Samiuela

Enter your first and last name separated by a comma
Frederick, Von Heimlen
Hello Von Heimlen, Frederick
```

USING COMMENTS

As a general rule, the overall program should have comments at the beginning, telling you what the program does. Each section (function) should also have comments explaining what it does and what values it returns. Any area in your program that is difficult to understand, less than obvious should be commented.

Listing . Demonstrates the use of comments

```
REM
'   Author:  Samiuela LV Taufa
'   Class:   Computer Studies 101
'   Date:    December 11, 1996
'
'   Purpose:
'   This is program shows user input, and computer output
PRINT "Enter your first and last name separated by a comma"
INPUT "", name$, lname$
PRINT "Hello "; lname$; ", "; name$
```

Qbasic lines that begin with the REM command are ignored, because REM stands for ‘remark’. The short-hand for typing in REM is to use the single quote (').

At the top of the Program File

Listed here are a number of good ideas for placing at the top of your program file. Which items you choose to include depends you’re your personal taste

- The name of the function or program
- What the function or program will do
- A description of how the program works
- The author’s name
- A revision history (notes on each change made)
- What compilers, linkers, and other tools were used to make this program
- Additional Notes

MATHEMATICAL EXPRESSIONS - CALCULATIONS

Variables are more interesting when you can manipulate or use their values for calculations. QBasic supports the standard mathematical functions such as multiplication, addition, subtraction and division.

The simplest example of using the mathematical operators is to use the PRINT statement such as:

```
cls
age = 25
PRINT 25 * 42
PRINT age * 42
```

This works very much like a simple calculator.

QBasic Operator Table

Operator	Sample	Explanation
=	A = B	Assigns value at its right to the variable at its left. Also used inside an IF statement to compare the value on the left to that on the right.
+	A + B	Adds value at its right to the value at its left.
-	A – B	Subtracts value at its right from the value at its left.
*	A * B	Multiplies value at its right by the value at its left.
/	A / B	Divides value at its left by the value at its right
\	A \ B	Integer division symbol. Same as divide, and also truncates result to an integer
^	A ^ 2	Exponentiation. Raises the number to its left to the power of the number on its right

Qbasic uses a table to distinguish how to represent mathematical symbols into the programming language. For example, since we cannot type in $y = x^2$ in Qbasic the mathematical method (looking at the table) is to type in y = x^2 (using the exponent operator) or we can use y = x * x (using the multiplier operator).

Given the following mathematical formulas, how would you write it in a line of QBasic programming?

Math Expression	QBasic Expression / Statement
$f(x) = y = x^2$	y = x * x, -or- y = x^2
f(x) y = (x + 2)(x + 1)	y = (x + 2) * (x + 1)
A = lw	A = l * w
k = 25 ÷ m	k = 25 / m

$A = \pi r^2$	pi = 3.141593
	A = pi * r * r
	A = pi * r ^ 2

C = 2π	pi = 3.141593
	C = 2 * pi * r

$f(x) = y = x^{½}$	y = x ^ (1/2)

Instructor Notes:

- Write up only the math expression, and work together with the class on the QBasic expression.
- Other mathematical functions known to the class would also serve as good examples for translating into QBasic Expression.

EXERCISE

To further evaluate, check, our understanding of how mathematical operators work, solve the following Qbasic formulas using the given values for the variables.

x	y	z	Formula	Value ?
4	11	3	z = x ^ 2 + y	z ?
2	1	3	y = 2 * x + 4 * z + y	y?
			y = 4*x^2 + 15*z + 32	y?

BEDMAS – ORDER OF EVALUATION

Formulas follow the standard arithmetic order of operation rules summarised by the term: BEDMAS

Level 1	**B**rackets
Level 2	**E**xponentiation
Level 3	**D**ivision and **M**ultiplication
Level 4	**A**ddition and **S**ubtraction

Higher levels are calculated before the levels below it. Level 1 (Brackets) have precedence, or are calculated before Level 2, 3, and 4. Likewise, Level 2 has higher priority and is calculated before Level 3, and 4.

Operations in the same level (for example Level 3, Multiplication and Division) can be performed in any order. 2 * 3 / 2 provides the same answer whether the multiplication is calculated first than the division, or the division is calculated first before the multiplication.

PROJECT EXERCISE

Write a program to calculate the volume of a circle by using the following formula, and values:

$V = \pi r^2 h$

$\pi = 3.141593$
$r = 10$
$h = 11$

PROBLEM SOLVING - THE PROBLEM RESOLUTION PROCESS

To help us understand one good process for solving a problem by using computer programming, let us try to solve the below problem using a five step procedure shown here:-

1 Determine the Purpose
2 What are the Required Data
3 Determine the Logic
4 Draft the Computer Program
5 Test & Re-test

CASE STUDY: FINDING THE CIRCUMFERENCE & AREA OF A CIRCLE

We have been asked to write a program to calculate the Circumference and Area of a circle. We have been given the following formulas for calculating these values.

Circumference: $C = 2\pi r$
Area: $A = \pi r^2$

1ST. DETERMINE THE PURPOSE

Read carefully the problem given, so we are sure we are solving the problem being asked. A serious problem is to spend a lot of time writing a program to solve a problem that no one wanted solved.

Ask questions if you are not sure you understand the problem. Ask questions to your teacher, your friends, the person who has the problem.

A good summary of the intended solution should be put into the documentation of your program.

Purpose: To calculate the Area & Circumference of a Circle.

The Sample Output.
It is often useful, where possible, to consider what the output of the program should look like. This helps the in the thinking, understanding of the process and purpose and design of the program, as well as the required data.

Sample Output

```
For a circle with radius __________
The Area is ____
The Circumference is _____
```

2ND. WHAT ARE THE REQUIRED DATA

What sort of information do we need to be able to solve our problem? Four useful categories for determining what data is required are:-

a What do we know ?
b What do we need to calculate
c What do we need to know ?
d What are the intermediary calculations?

(a) What do we know ?

- From the problem statement we know the mathematical formulae required to calculate the values.
- We know π is a really big number that we can approximate by using a smaller number such as 3.141593

(b) What do we need to calculate ?

- We need to calculate the *Circumference* and the *Area*
- We know the formula for this calculation in Mathematical Terms

Circumference: $C = 2\pi r$
Area: $A = \pi r^2$

- We know from our previous exercises, how to convert the mathematical formula into the QBasic Statement

Problems not easily resolved with a simple mathematical calculation may require you to think through differing sequences of calculations, instead of a single formula.

(c) What do we need to know ?

- The formula requires the *radius*

(d) What are the intermediary calculations ?

There are many problems where we are sometimes required to make calculations of other values before we can actually isolate and explain what the answer to the specified problem is.

- For this problem we do not have to make *intermediary* calculations.

3^{RD}. DETERMINE THE LOGIC

The *logic of a program*, often called the *algorithm*, is the sequence of instructions required to create the solution, answer, to the problem.

Although it may seem difficult to determine the logic or the sequence to solving a problem, it often is simplified by starting the thinking process instead of worrying about whether it can be done or not.

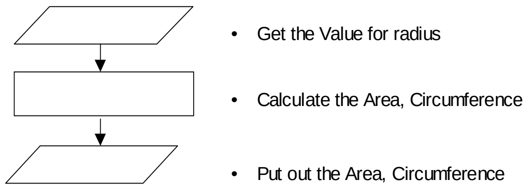
Sample Logic/Algorithm Design Process:

Attempt 1.

- Calculate the value of the *Area* and *Circumference*
- *oops* we don't know what the *radius* may be.

Attempt 2.

- Get the value for *radius* from somewhere (lets get it from the user)


• Get the Value for radius
• Calculate the Area, Circumference
• Put out the Area, Circumference

- Calculate the value of the *Area* and *Circumference*

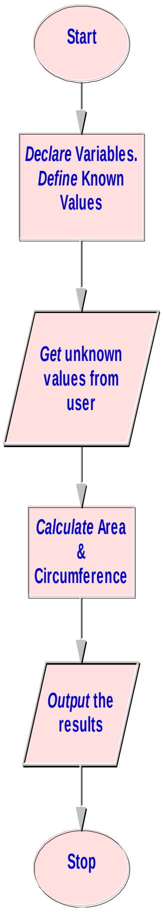
- Put it out for the user to see
- *clean up the logic process*

Attempt 3.

- *Start*
- *Define* known values (eg. π)
- *Get* the required unknown values, *radius*
- *Calculate* required values, *Area and Circumference*
- *Output* the data for the user
- *Stop*

Some House Cleaning

It is good practice at this point to define the variable names to be used. Again, use variable names that are self-explanatory, so lets define the variables we will use:



Start
Declare Variables. Define Known Values
Get unknown values from user
Calculate Area & Circumference
Output the results
Stop

(a)

(b) Declare *Circumf, Area, PI, radius. We are also define PI to be 3.147*

(c) *radius*

(d) *Area, Circumf*

(e) *Area, Circumf*

(f)

Circumf for the Circumference C, may contain decimal values so we use either a SINGLE or DOUBLE.

Area for the Area A, may contain decimal values so we use either a SINGLE or DOUBLE. and

PI for the value of π, may contain decimal values so we use either a SINGLE or DOUBLE.

radius will be used for the value of the radius which we will ask from the user, may contain decimal values so we use either a SINGLE or DOUBLE.

4TH. DRAFT THE COMPUTER PROGRAM

Although some people may think their programming skills are cool enough that they are now ready to create the program on the computer, it is always good practice to draft your program on paper, and to test the logic implementation by paper-execution of your program.

This is the time where we translate our logic/algorithm designed in the previous step, into the computer instructions specific to the programming language we are using (QBasic)

5TH. TEST AND RE-TEST

Many programmers become confident (*arrogant)* of their skills to the point they neglect to adequately test their programs. The worst thing for a user is to trust a program will work correctly and have it fail because the programmer couldn't be bothered to spend enough time working through the program to minimise potential errors.

Spend as much time as possible testing, and re-testing your programs.

For our exercise: We know that the only value that changes is *radius* so it is a good idea to calculate on paper (or using your calculator) a few values for *radius* and then compare this with the program result.

Listing - Suggested Solution

```
REM
'   Author:
'   Class:
'   Date:
'
'   Purpose:
'   This program will calculate the Area and Circumference of a Circle
'   after recieving the value for the radius from the user.
'
'             (a) Start
CLS
'             (b) Declare Variables, Define Known Values
DIM radius AS DOUBLE   ‘ Radius
DIM PI AS DOUBLE       ‘ pi
DIM Circumf AS DOUBLE ‘ Circumference
DIM Area AS DOUBLE     ‘ Area
PI = 3.141593

COLOR 9
'             (c) Get Required data from user
PRINT “This program calculates the Area and Circumference of a Circle”
INPUT “What is the radius of the circle”; radius

'             (d) Make the Calculations
Circumf = 2 * PI * radius
Area = PI * radius ^ 2

'             (e) Send the results to the screen for the user
COLOR 2
PRINT "For a Circle of radius"; radius;
PRINT "and using a value for PI as"; PI
PRINT "The calculated Circumference is"; Circumf
PRINT "The calculated Area is"; Area
'
'             (f) Stop
```

Values for radius,	Manual Calculation	Program Calculation
12	....	....
14	....	....
15	....	....
22	....	....
45	....	....

FLOW CONTROL

Program flow is the order, or sequence (*flow*) from one program instruction to another. We have so far seen program flow sequentially from the first instruction to the last. Flow control are the methods for changing, controlling the flow by which instructions are 'executed' or 'processed' by the computer.

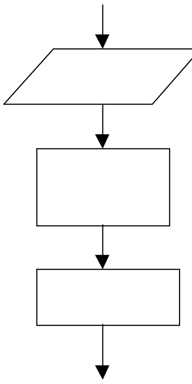


Fig. Linear Program Flow

Linear. When program instructions are processed one after the other is a straight sequence, this flow is called "linear". In our flow-chart symbols, we can see this by the arrows always flowing straight down without any deviations.

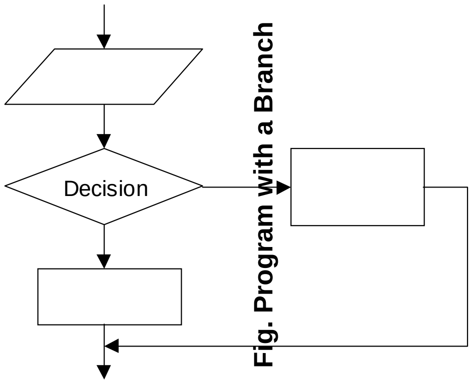


Fig. Program with a Branch

Branching. When a program execution changes direction for only a specified condition, this is called branching. We can see this in our flow-chart symbols when certain instructions may possibly be ignored, or when certain instructions may only be accessed for a given condition. Whenever a change in instruction flow is diagrammed it is always shown as a diamond shape, indicating that the program makes a decision at that point for which side it will branch to. We will be taking a further look at branching in this section.

Branches are designed to allow certain instructions to be used, while ignoring others. Branches can also be used to ignore certain instructions if the conditions are not right for using them.

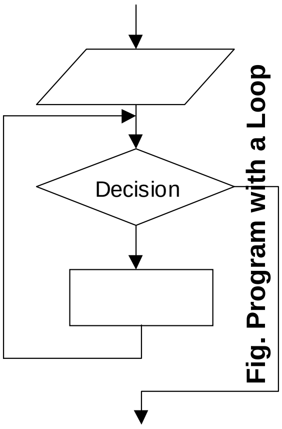


Fig. Program with a Loop

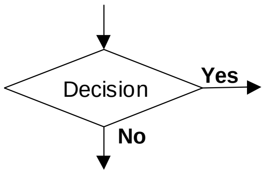
Loops, iterations. When a sequence of instructions need to be repeated, then we are performing a loop. Repeating a set of instructions is controlled by specifying a decision for when the loop should be repeated, and when it is time to leave the loop.

Diamond – Decision. The key, and power in branching is making a decision to do things differently. Remember that this is the special facility which differentiates computers from other tools, the ability to act make a decision dependent on the data presented to it.

Understanding of this Unit is very useful for developing students diagnosis skills, and in other course modules: database queries, spreadsheet what-if scenarios.

To simplify the diamond decision, the decision can either return a *"yes"* or *"no"*. If the decision results in a Yes, then we continue with the *"yes"* activities, otherwise we look at what is to be done when the decision is a no.

Reviewing the branching and loops, we can more clearly see the difference the decision can make in choosing which parts of a program to continue with.



Branching. In our flow-chart diagram, when we reach the decision two things can happen, either our decision results in a *“yes,”* or a *“no”*.

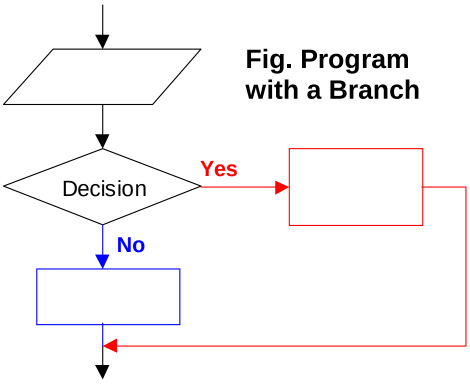


Fig. Program with a Branch

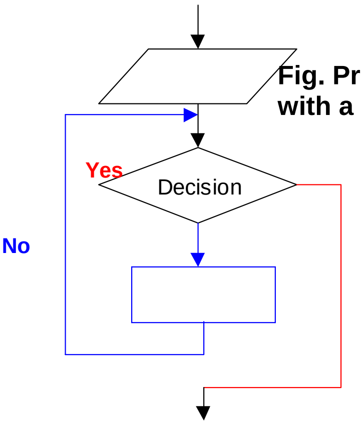


Fig. Program with a Loop

- *“yes,”* we ‘branch’ to the red process complete that exercise, skip the blue box and continue the program.
- *“no,”* we ‘branch’ to the blue process complete that exercise, skip the red box and continue the program

Loops, iterations. In our flow-chart diagram when we reach the decision two things can happen, either our decision results in a *“yes,”* or a *“no”*.

- *“yes,”* we skip the blue box and continue the program.
- *“no,”* we complete blue process and at the end of the blue process we come back to check the decision again.

Enough problems require our programs to make decisions about different things to do that it is not sufficient to solve problems using simple mathematical formulas.

Preamble

Before we can discussion decision making processes (keywords and methods used by QBasic to make comparisons) we need to first of all understand how comparisons are made. The following section is a discussion of how comparisons are made in computer programming.

Decision Making – Comparisons

Looking at Relationships and Logic

Decisions are made through comparisons. One comparison is to compare the relationship between two values, another comparison is to compare different relationships.

Relational Operators – Mathematically

Mathematical relationships compare the value of the variable on the *left hand side* of the relationship *operator* to the value of the variable on the *right hand side*. The following Relationship Operations table summarises the operator use.
For example; if we have three variables a%=1, b%=2, c%=3 then the relationship in the example below is clear.

Table: Relationship Operations

Operator	Sample	Relationship	Result
=	5 = 3	Equality. Compares the value on the left whether it is equal to the value on its right.	No
>	5 > 3	Greater than. Compares the value on the left whether it is greater than the value on its right	Yes
<	5 < 3	Less than. Compares the value on the left whether it is less than the value on its right	No
<>	5 <> 3	Not Equal To. Compares the value on the left whether it is not equal to the value on its right	Yes
>=	5 >= 3	Greater than or Equal To.	Yes
<=	5 <= 3	Less Than or Equal To.	No

Linking the comparison again with our flow-chart diagrams, one line of code will look like the following chart.

```
a%=1: b%=2: c%=3

a% < c%                  ' is a% < c% ? YES
c% >= a% + b%            ' is c% >= a% + b% ? ___
c% - b% = a%             ' is c% - b% = a% ? ___

c% < a                   ' is c% < a ? NO
b% = a% * 3              ' is b% = a% * 3 ? ___
b% >= c%                 ' is b% >= c% ? ___
```

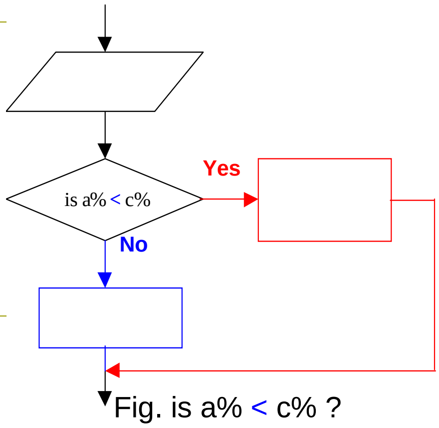


Fig. is a% < c% ?

Our decision, condition, is if a% < c%, *"yes"* then we will branch and complete the red, while if the decision is *"no"* then we will complete the blue branch. A general term for the diamond, or decision, is that it is a condition. Depending on the 'condition' different parts of the program will be processed.

PROGRAM SAMPLE

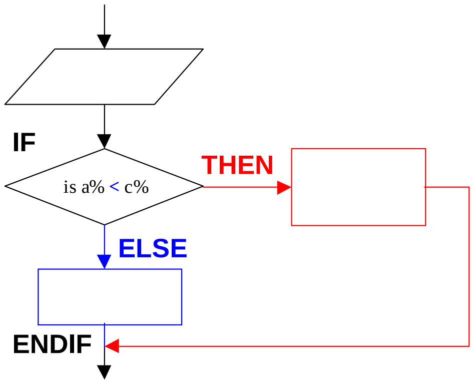


Fig. is a% < c% ?

Translating the flow-chart into Qbasic programming code, we use the template (outline) which is described in more detail later.

IF *condition* THEN
  stuff in red
ELSE
  stuff in blue
ENDIF

If we stick the IF, THEN, ELSE into the diagram. We have the **IF** marking our decision. **THEN** is what happens when the decision results in a *"yes"* and **ELSE** is what happens when the decision results in a *"yes"*. **ENDIF** marks where things go back to the normal sequence of program instructions.

Using the above structure (template) we now have a program code that will look something like the following:

PROGRAM EXERCISE

```
IF a% < c% THEN
   stuff in red
ELSE
   stuff in blue
ENDIF
```

Using the above example, and the template, write the program code for the other six decision examples, such as c% >= a% + b%.

ORDER OF EXECUTION – WHICH DO WE DO FIRST ?

When an expression contains a comparison and arithmetic operations *(such as addition, subtraction, etc.)* QBasic will perform the arithmetic before comparing the results.

COMPARING COMPARISONS

When a problem requires using more than one relationship, it becomes difficult to keep track of what is happening to the code when the problem requires checking multiple relationships. Comparing comparisons evaluates more than one relationship, as one comparison, and makes a decision whether it is a yes or a no.

When a comparison statement *(expression)* evaluates more than one relationship then we use the *logical operators* AND, OR, NOT to compare the logic.

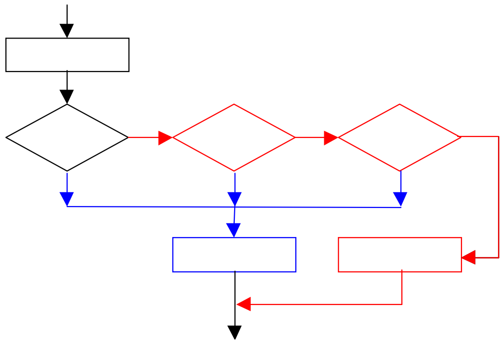


Fig. Comparing different relationships can get messy

AND. When we use AND to combine two relationships, then both relationships have to be TRUE for the AND operator to return a TRUE. When either one is FALSE, then AND will return a FALSE.

OR. When at least one of the relationships is TRUE, the OR will return TRUE. When both relationships are FALSE then OR will also return a FALSE.

Expression1	Expression2	AND	OR
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F
The Logical Condition Comparisons			

NOT. When the relationship is TRUE, **not** will reverse it and return a FALSE. When the relationship is FALSE, **not** will reverse it and return a TRUE.

Expression	NOT
T	F
F	T
The Logical Condition Comparisons	

Continuing the previous example:

The Table: *Logical Condition Comparison* summarises the above discussion when comparing two separate test expressions, *expression1* and *expression2*

```
a%=1: b%=2: c%=3

a% < c% AND c% >= a% + b%             ' TRUE AND TRUE —> TRUE
c% >= a% + b% AND c% < a              ' TRUE AND FALSE —> FALSE
c% < a AND b% = a% * 3                ' FALSE AND FALSE —> FALSE

c% >= a% + b% OR c% < a               ' TRUE OR FALSE —> TRUE
c% < a OR b% = a% * 3                 ' FALSE OR FALSE —> FALSE

NOT (c% - b% = a%)                    ' NOT (TRUE) —> FALSE
NOT (b% >= c%)                        ' NOT (FALSE) —> TRUE
```

PROGRAM SAMPLE

Translating the a% < c% AND c% >= a% + b% sample into Qbasic programming code, we refer to the template (outline).

IF *condition* THEN
  stuff in red
ELSE
  stuff in blue
ENDIF

```
IF a% < c% AND c% >= a% + b% THEN
   stuff in red
ELSE
   stuff in blue
ENDIF
```

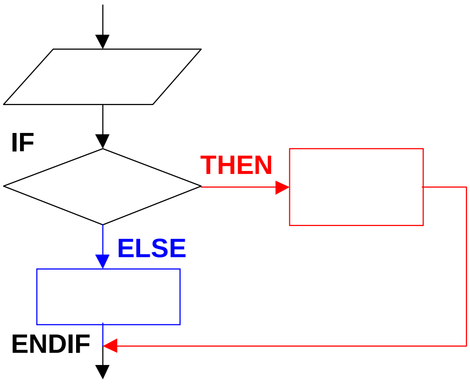


Fig. Comparing Comparisons ?

Translating the NOT (c% - b% = a%) sample above into Qbasic programming code, we refer to the template (outline).

```
IF NOT (c% - b% = a%) THEN
   stuff in red
ELSE
   stuff in blue
ENDIF
```

PROGRAM EXERCISE

Using the above example, and the template, write the program code for the other six decision examples, such as c% >= a% + b% OR c% < a.

ORDER OF EXECUTION.

When an expression contains logical operators, relational operators and arithmetic, then QBasic evaluates the expression in the following sequence.

- arithmetic operations are performed first
- relational operations are compared from left to right
- logical operations are performed in the order (NOT, AND, OR)
- brackets can be used to over-ride the default order.

THE COMPLEX AND THE COMPOUND

Queries, or decisions made when combining more than one test expression is sometimes called a "Compound" decision.

DECISION MAKING – THE IF STATEMENT

The simplest flow control statement is the IF conditional statement. The IF statement provides your program with the ability to make decisions about when it is appropriate to execute different portions of the instructions.

```
IF condition1 THEN
     [statementblock-1]
[ELSEIF condition2 THEN
     [statementblock-2]]...
[ELSE
     [statementblock-n]]
END IF
```

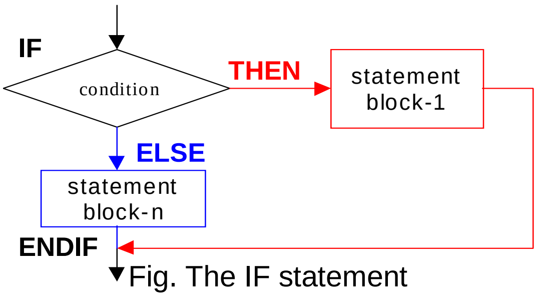


Fig. The IF statement

Instructor Notes:
It is helpful in this section to 'mark' for students the statement blocks surrounding the IF statement.

First the *condition1* is evaluated. If it is *true*, the statements contained in the statementblock-1 are executed. After that is executed, then the program continues after the END IF line. If not, the statements contained in the ELSEIF block are evaluated, executed.

Both the ELSEIF and ELSE statements are optional, and either both can be absent or both can be in the program.

A simple example

We want to check a number, variable numb% whether it is a positive number or not. So we give it one try.

```
IF numb% > 0 THEN
     PRINT "This number is a positive number"
END IF
```

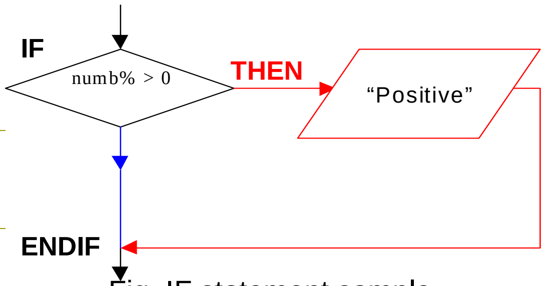


Fig. IF statement sample

First the *condition* is evaluated. If *numb%* > 0, is YES then the PRINT statement is executed. After that is finished, then we continue with the next line after the END IF line.

If the condition is NO, then we continue to the next line in the program.

```
IF numb% > 0 THEN
     PRINT "This number is a positive number"
ELSE
     PRINT "This number is negative or zero"
END IF
```

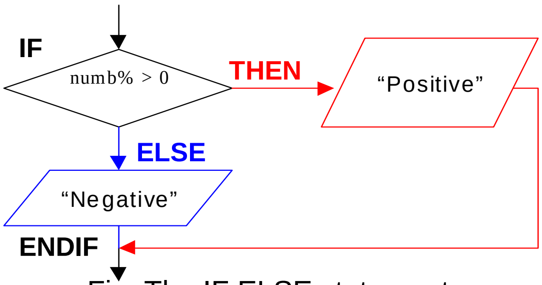


Fig. The IF ELSE statement

Teaching Note:
The following pages provide more specific examples of the IF … END IF structure. To practise and verify students understanding of this structure it is suggested that at least two or more examples (of simply the decision structure) be discussed and reviewed during class time. The review should include the drawing and flow of flow-chart diagrams.

Two samples of source is provided here.

```
IF age% > 22 THEN
   PRINT "Too Old ... "; age%
END IF
```

```
IF age% < 5 THEN
   PRINT "Little Baby"; age%
END IF
```

CASE STUDY. CHECKING THE SALES RESULTS FROM THE CANTEEN

Our school canteen sometimes makes money, sometimes it does not make as much money as we want. We would like a computer program that will compare the daily sales to the Canteen goal sales.

We can set up a *target*, and ask for the *sales.* When the sales meets or exceeds the target then we are happy, but when the sales are below the target this is not a good thing.

Determine the Purpose.

Stated above.

What are the Required Data.

- Target Sales
- Canteen Sales

Determine the Logic.

- When sales meets or exceeds target, we are happy.

Draft the Computer Program.

Test & Re-test.

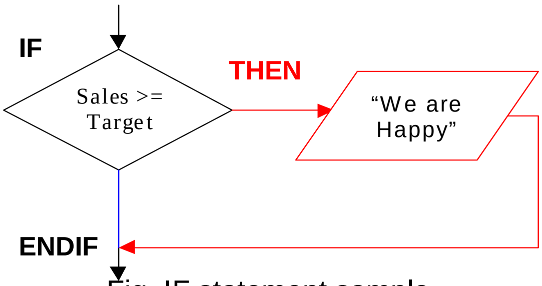


Fig. IF statement sample

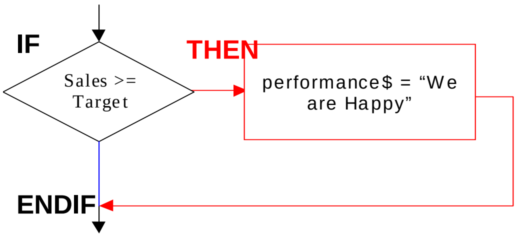
PROGRAM EXERCISE

Create a flow-chart of the suggested solution as described above.

Instructor Notes: Review the flow-chart and a selected solution with students.

The following is a sample solution to the stated problem.

```
performance$ = "We are not happy"     ' We are starting off thinking this is
INPUT "Target       "; target         ' not going to be good.
INPUT "canteenSales "; canteenSales

IF canteenSales >= target THEN
   performance$ = "We are Happy"
ENDIF
PRINT "*===========================*"
PRINT "Target "; target
PRINT "Sales "; canteenSales
PRINT "Performance "; performance$
```



CASE STUDY. PROVIDING A BONUS WHEN SALES ARE HIGH

To give our canteen staff incentives to try and increase sales we decided that we will give them a bonus if the sales is higher than the target. This requires an extra decision and work path from the previous problem.

When sales exceeds or meets the target we want to give a bonus, but when the sales are below the target we want to not give a bonus.

Determine the Purpose.

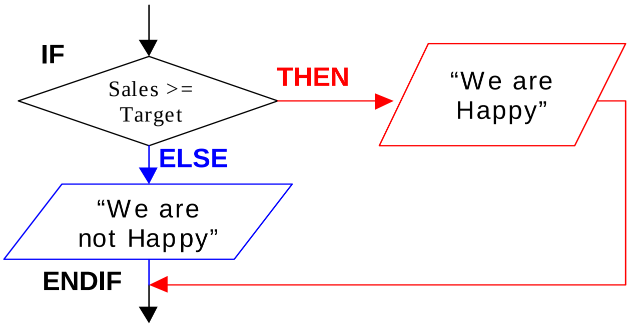
Stated above.

What are the Required Data.

- Target Sales
- Canteen Sales
- Bonus Amounts

Determine the Logic.

- When sales meets or exceeds target, we are happy, and we give a bonus.
- When sales is less than target, we are not happy, and we give no bonus.



Draft the Computer Program.

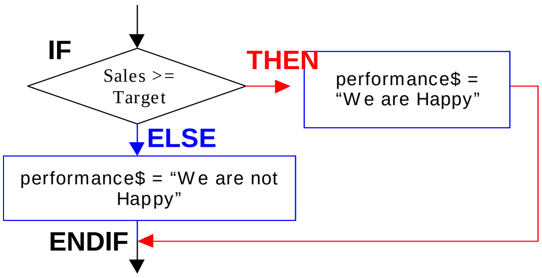
Test & Re-test.

PROGRAM EXERCISE

Create a flow-chart of the suggested solution as described above.

The following is a sample solution to the stated problem.

```
INPUT "Target       "; target
INPUT "canteenSales "; canteenSales

IF canteenSales >= target THEN
   performance$ = "We are Happy"
   bonus = 100 + 0.01 * (canteenSales - target)
ELSE
   performance$ = "We are NOT Happy"
   bonus = 0
ENDIF
PRINT "*==========================*"
PRINT "Target "; target
PRINT "Sales "; canteenSales
PRINT "Performance "; performance$
PRINT "Bonus "; bonus
```



IF
Sales >= Target
THEN
performance$ = "We are Happy"
ELSE
performance$ = "We are not Happy"
ENDIF

CASE STUDY. PROVIDING DIFFERENT LEVELS OF BONUSES

We find that the incentive program works really well, so management would like us to refine provide different levels of performance incentives depending on how much the sales is above the *target*.

Determine the Purpose.

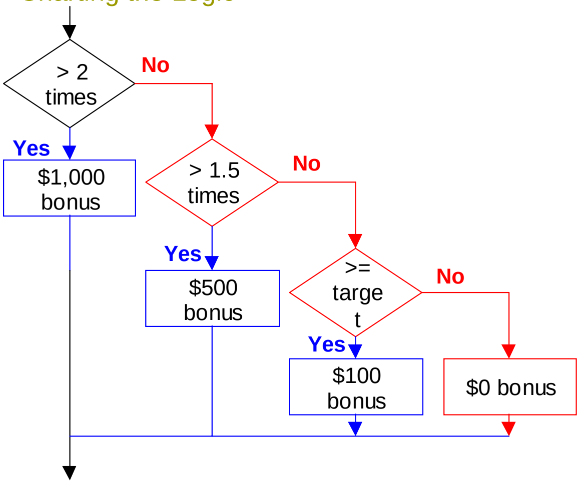
Stated above.

What are the Required Data.

- Target Sales
- Canteen Sales
- Bonus Structure

Determine the Logic.

Charting the Logic



- When sales meets or exceeds target, we are happy, and we give a bonus depending on a 'formula' given to use by management:
  - When sales is twice or more the target, give a bonus 1000
  - When sales is One and a half times target, give a bonus 500
  - When sales is greater than the target, give a bonus 100
  - When sales is below the target, we need new people.
  - otherwise, the canteen staff are sacked.
- When sales is less than target, we are not happy, and we give no bonus.

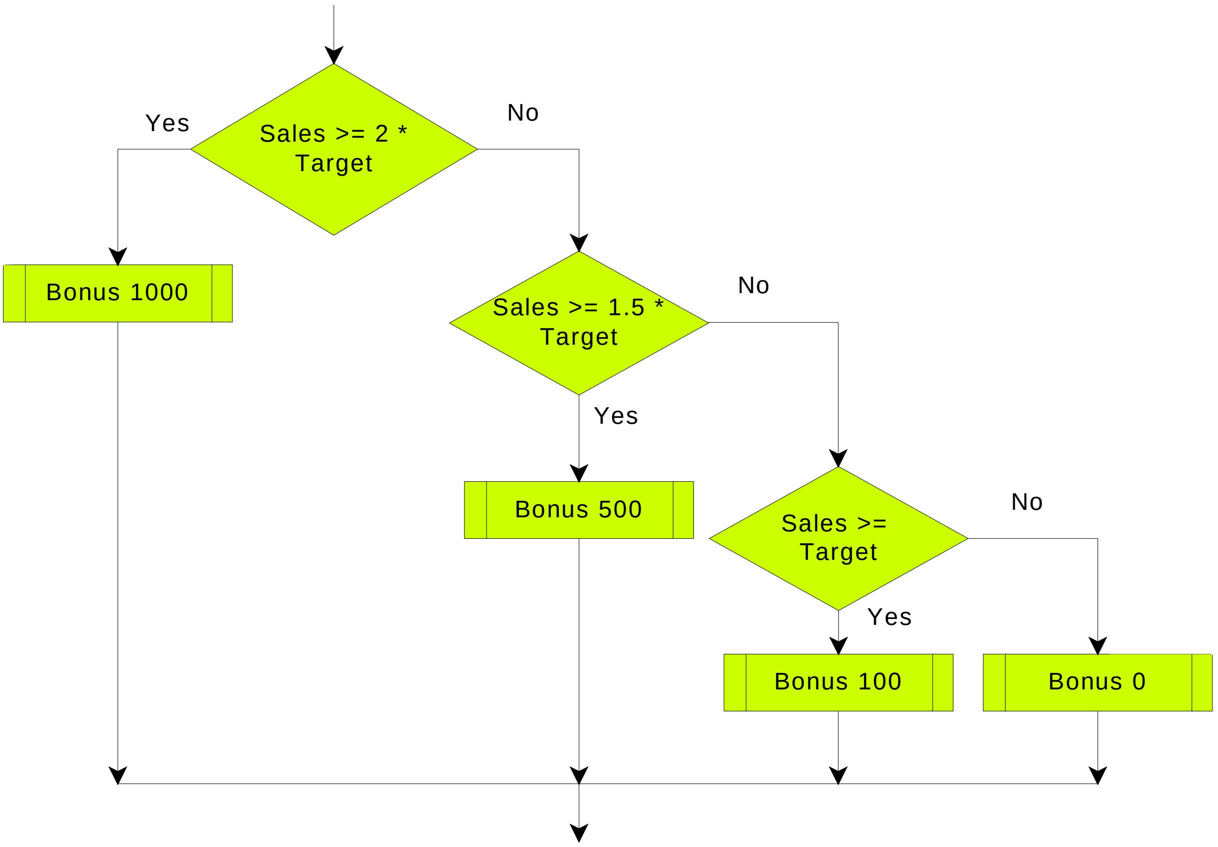
Draft the Computer Program.

- Mathematical representation of the Bonus 'formula'.
  - >= 2 * target; bonus = 1000
  - >= 1.5 * target; bonus = 500
  - >= target; bonus = 100
  - Otherwise bonus = 0; good-bye.

Test & Re-test.

PROGRAM EXERCISE

Create a flow-chart of the suggested solution incorporating the mathematical representation described above.



The following is a sample solution to the stated problem.

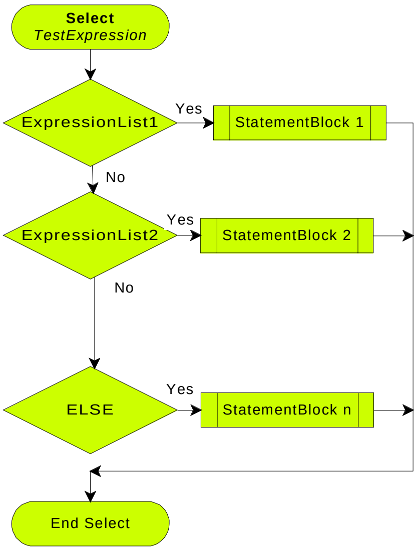
```
INPUT "Target       "; target
INPUT "Sales "; canteenSales

IF canteenSales >= 2 * target THEN
   performance$ = "Excellent"
   bonus = 1000
ELSEIF canteenSales >= 1.5 * target THEN
      performance$ = "Fine"
      bonus = 500
   ELSEIF canteenSales >= target THEN
      performance$ = "Satisfactory"
      bonus = 100
   ELSE
      performance$ = "Unsatisfactory"
      PRINT "You're FIRED !!!!"
ENDIF

PRINT "Target "; target
PRINT "Sales "; canteenSales
PRINT "Performance "; performance$
PRINT "Bonus "; bonus
```

Decision Making – The SELECT CASE construct

The IF condition is nice, but it is difficult to follow when more and more comparisons are required. The *IF – ELSE* flow control makes it easy to write programs to choose between *two* alternatives, however, a program needs to choose one of *several* alternatives. We can do this by using *IF .. ELSE IF .. ELSE .. END IF*, but in many cases it is more convenient to use the *SELECT CASE* flow control statement.

SELECT CASE allows you to write program code that is easier to interpret/read and therefore easier to correct, de-bug. The SELECT CASE provides a better program structure for multiple decisions/alternatives.

```
SELECT CASE testexpression
  CASE expressionlist1
    [statementblock-1]
  [CASE expressionlist2
    [statementblock-2]]...
  [CASE ELSE
    [statementblock-n]]
END SELECT
```



The SELECT statement uses the value of *testexpression* to transfer control to one of the CASE statements for execution. The *expressionlist* is a list of values which is compared to the testexpression to determine which CASE statement is executed. When the *testexpression* does not match any of the *expressionlists*, then the CASE ELSE statement is executed.

CASE STUDY. DAY OF THE WEEK

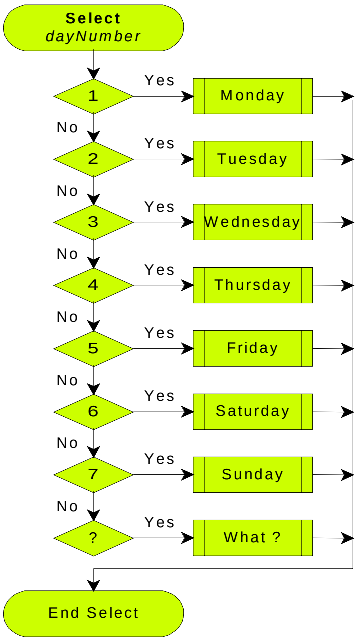
We know what number of day in the week it is, but is that Monday, Tuesday, or

SELECT DAYS

```
CLS
INPUT "Date Number "; dayNumber

SELECT CASE dayNumber
  CASE 1
     PRINT "Monday"
  CASE 2
     PRINT "Tuesday"
  CASE 3
     PRINT "Wednesday"
  CASE 4
     PRINT "Thursday"
  CASE 5
     PRINT "Friday"
  CASE 6
     PRINT "Saturday"
  CASE 7
     PRINT "Sunday"
  CASE ELSE
     PRINT "How many days in your Week?"
END SELECT
```

Note:
For improved student learning; It is important that this OUTPUT exercise be completed together on the board with students.

Wednesday ?



In this example, the SELECT CASE *testexpression* uses the variable *dayNumber* to determine which CASE statement will be executed by the SELECT CASE.

The SELECT CASE checks the *testexpression* which in our above example is the value in the variable *dayNumber*. Then the program scans the list of "*expressionlists*" until it finds one that matches that value. The program then jumps to the statement block in that CASE.

What if there is no match? If there is no match, then the program looks for the CASE ELSE statement – block. If there is no match, and no CASE ELSE then the program goes to the end of the SELECT CASE block and continues with the next program statement.

In Class Exercise

1. Draw a flow-chart of the Select Case program

2. Run a desk-check using the values 3 and then 8 and write down what the expected result is going to be.

Summary

The SELECT CASE checks the *testexpression* then the program scans the list of "*expressionlists*" until it finds one that matches that value. The program then jumps to the statement block in that CASE.

If there is no match then the program goes to the end of the SELECT CASE block and continues with the next program statement.

In Class Exercise

1. What is the variable used as the *testexpression*

2. Which statement block will the program flow to if the *testexpression* is 3 ?

3. Which statement block will the program flow to if the *testexpression* is 0 ?

4. What number does the user have to enter to get the message about Prefect ?

Group Exercise

1. Draw a flow-chart of the select case to place into the variable mth$ the actual month name (eg. January, February) if we input the value of the month into the variable mth_numb%

2. Write the program code to match the above created flow-chart.

CASE STUDY. WHICH SCHOOL YEAR ARE WE IN, IN TONGA

Given a student's Form year (in one of Tonga's Secondary Schools) what status is that student ?

1st year, still a babe
2nd – 4th year, developing slowly
5th year and greater, good material.

In Class Exercise

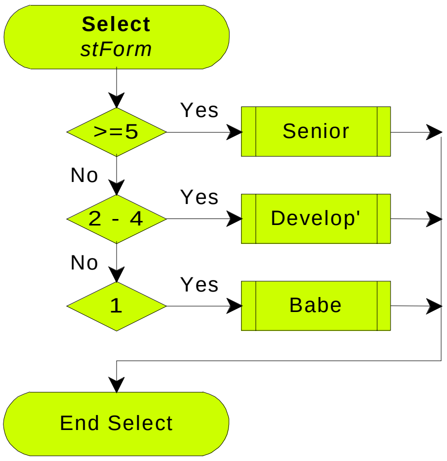
1. Draw a flow-chart of the select case to display a message related to the input the value of the student Form in the variable stForm%

2. Write the program code to match the above created flow-chart.

Individual Exercise

1. Draw a flow-chart of the select case to place into the variable status$ the actual student status (ie. Freshman if 1st year student, Sophomore if 2nd year student, Junior if 3rd year student, Senior if 4th year student) if we input the value for the student year into the variable stud_year%

2. Write the program code to match the above created flow-chart.



The following is a sample solution to the stated problem.

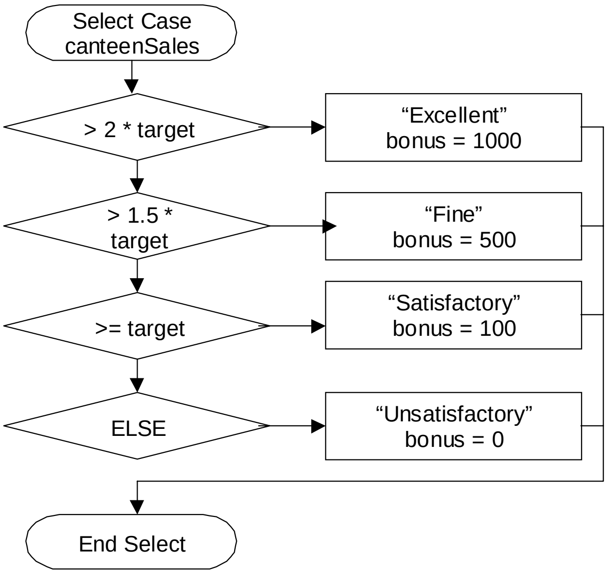
SELECT STUDENT CLASS

```
CLS
INPUT "Which Form is the Student in (1-6): ", stForm

SELECT CASE stForm
  CASE IS >= 5
   PRINT "A Senior student eligible for Prefect selection"
   PRINT "Vakai'i atu pe 'oku fie mataa-pule pe 'ikai."
  CASE 2 TO 4
   PRINT "Developing Slowly."
   PRINT "Ko e fa'ahinga taimi faingata'a, pe 'e tokanga ki he ako, pe ?"
  CASE 1
   PRINT "Just a Jelly-bean."
   PRINT "Tokanga'i hono tauhi 'etau Babe."
END SELECT
```

Revisiting the Canteen Sales Program

Shown below is a flow-chart diagram and rewrite of the Canteen Sales bonus incentive using the SELECT CASE structure.

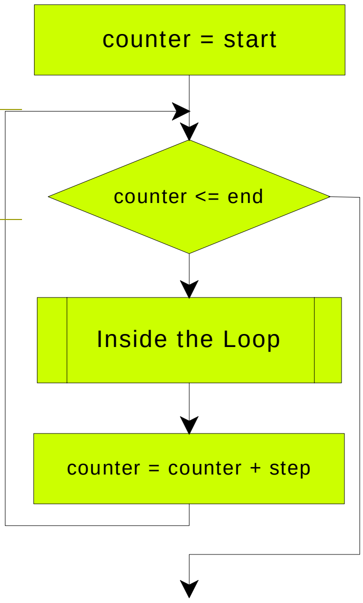
```
'
' Another look at the IF3 program us-
               ing SELECT CASE
'
CLS
INPUT "Target       "; target
INPUT "Sales "; canteenSales

SELECT CASE canteenSales
CASE IS >= 2 * target
       performance$ = "Excellent"
       bonus = 1000
CASE IS >= 1.5 * target
       performance$ = "Fine"
       bonus = 500
CASE IS >= target
       performance$ = "Satisfactory"
       bonus = 100
CASE ELSE
       performance$ = "Unsatisfactory"
       bonus = 0
       PRINT "You are FIRED!!"
END SELECT

PRINT "Target "; target
PRINT "Sales "; canteenSales
PRINT "Performance ";performance$
PRINT "Bonus "; bonus
```



The above sample shows how much easier it is to follow the program flow in a select statement over the IF THEN ELSE when more than two relation comparisons have to be used.

REPETITIONS – THE FOR LOOP

The most common of the loops is the FOR loop. Loops are programming structures to allow the programmer to repeat a number of instructions. The FOR loop is the simplest because the number of times the ‘loop’ is repeated is always determined before executing any of the statements inside the loop. Loops are also called *iteration* structures.



The FOR loop structure has the following appearance.

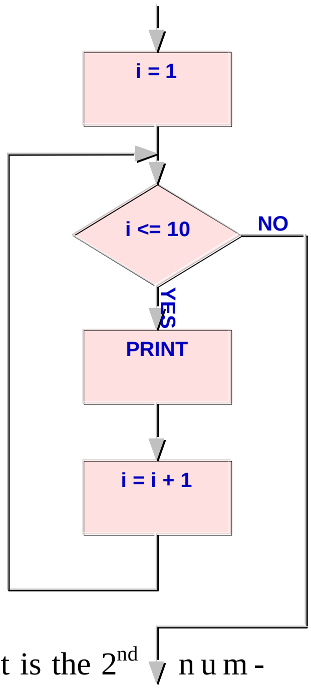
```
FOR counter = start TO end [STEP increment]
    [statementblock]
NEXT [counter [,counter]...]
```

1. The variable *counter* is assigned the value of *start*.
2. The value of *counter* is compared to the value of *end.*

* if the value of counter is less than end, the statementblock is executed.

* if the value of counter is not less than end, the FOR ... NEXT is completed and the next statement to be executed is the statement following the FOR ... NEXT.

A simple example of how the FOR loop works follows



```
FOR i = 1 TO 10
   PRINT “This is the “; i; “time ...”
NEXT i
```

In this loop we have chosen ‘*i*’ as the variable name to be our *loop counter*.

The loop demonstrates going through the numbers 1 through to 10 {ie. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. The loop/iteration repeats the statements inside the loop 10 separate times. In each repetition, the counter *i* is assigned a different value according to the list of numbers between 1 and 10.

LOOP PROCESS

i values is the set {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

- The 1st time through the repetition; 1 is put into *i* because it is the 1st number in the list.
- The second time through the loop; 2 is put into *i* because it is the 2nd number in the list, etc.
- This continues until the last number from the list is used.

Inside the Loop.

There is only one statement to be processed (executed), a PRINT command to send output to the screen. This command is repeated 1 through 10 times, and each time it will output the value in the *counter variable i*.

The For loop structure is very good for activities that require a predetermined

Output

```
This is the 1 time ...
This is the 2 time ...
This is the 3 time ...
…
This is the 10 time ...
```

repetition. Predetermined: If before the loop starts we know exactly how often the loop will be repeated, then we can say that loop has a ‘*predetermined*’ repetition.

CASE STUDY. LOOPING THROUGH THE COLOURS

We are told that QBasic supports 0 to 15 colours for the display of text with the PRINT command. (Remember Americans spell it COLOR?)

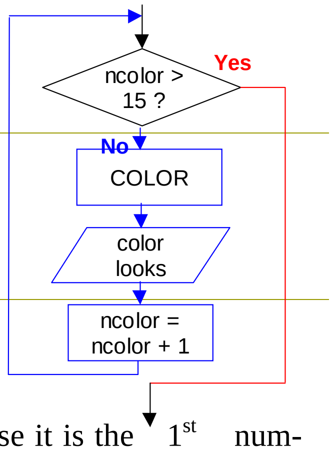
COLOR
For those who wish to know more about the COLOR statement, more information is available in the Qbasic help screens.

We do not know what these QBasic colours look like, and we could write 0 to 15 different lines of code for each color, but lets see if we can use the new loop structures.

1 We need to go through the numbers 0 through to 15 so we can use a FOR counter = 0 to 15.

2 Let us use *ncolor* as the counter variable

Loop Process



Listing

```
CLS
FOR ncolor = 0 TO 15
   COLOR ncolor
   PRINT "Color "; ncolor; "looks like this"
NEXT ncolor
```

ncolor values is the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

- The 1st time through the repetition; 0 is put into *i* because it is the 1st number in the list.
- The second time through the loop; 1 is put into *i* because it is the 2nd number in the list, etc.
- This continues until the last number from the list is used.

Inside the Loop.

Each program statement is executed from the beginning of the loop to the end.

QBasic's Colors	
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
White	7
Grey	8
Light Blue	9
Light Green	10
Light Cyan	11
Light Red	12
Light Magenta	13
Yellow	14
Very Bright White	15

Output

```
Color 0 looks like this
Color 1 looks like this
Color 2 looks like this
…
Color 15 looks like this
```

CASE STUDY. CALCULATING THE SUM OF NUMBERS:

We want to use the computer as a calculator to be able to calculate the sum of numbers between 1 and any given number *n* we enter into the computer.

Determine the Purpose.

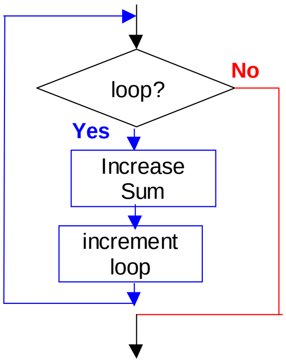
Stated Above.

What are the Required Data.

- We need a number *n*
- We need to calculate the sum.

Determine the Logic/Algorithm.

- We need a variable to store the sum (*sum*)
- We need to go through the numbers from 1 to the actual number *n* and for each number add it to *sum*.
  - The set is *{1, 2, 3, …, n}*
  - Add 1 to *sum*
  - Add 2 to *sum*
  - Add 3 to *sum*
  - ..
  - Add *n* to *sum*
- When we finish going through the list of numbers, then we are finished.

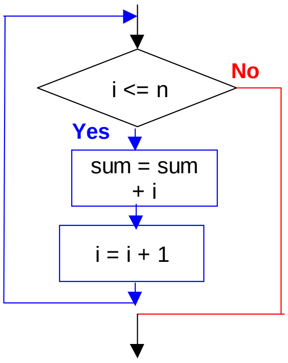


1st Attempt at algorithm

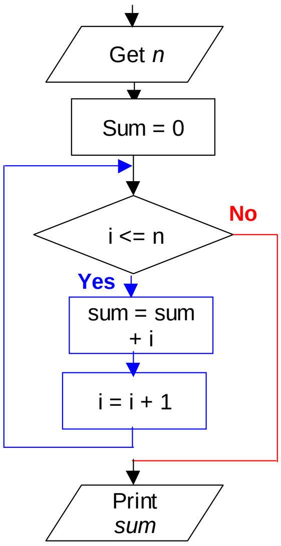
Draft the Computer Program.

- The above displayed logic shows a consistent repetition of the Add *number* to *sum*. This leads us to review whether a loop is the best method for performing this task.
- Before beginning the loop, we will know exactly how often the loop is to be performed (from the user input)
- Use *n* as the number the user will enter.
- Use *sum* to hold the total of the numbers. The total is initially 0.
- Use *i* as the loop counter, the set is *{1, 2, 3, …, n}*

  FOR i = 1 to n
  - Use *sum = sum + loop_counter* as the formula for calculations



2nd Attempt at algorithm



Extending the 1st two flow-charts, we get a flow chart similar to the one on the left.

1. We need to get the number *n* we want to work with. Let us get this from the user.
2. We set the current *sum* to 0 (zero) since we haven't started counting.
3. We set up the loop (using our previous chart)
   - Check if we still have a valid number to add ?
   - add the loop counter to sum
   - increment, add 1, to the loop counter
4. Print out our calculated *sum*

Note:
For improved student learning; It is important that this OUTPUT exercise be completed together on the board with students.

Sample Solution – Finding the sum of earlier numbers

```
PRINT "This program calculates the sum of numbers between 1 and"
PRINT "any positive number you enter."
INPUT "What is the number to be summed"; n

Sum = 0              ' The initial total is zero
FOR i = 1 TO n
  Sum = Sum + i
NEXT i
PRINT "The sum of 1 + ... + n (when n is > 1) is"; Sum
```

Test & Re-test

1. Make sample calculations on paper and calculator, and compare these to the computer program calculated results.

In Class Exercise

1. As in the previous Loop examples, write down the set and values for the loop counter when the user enters 5 for *n*.

2. Write down what you analyse will be the output for the above program when the user enters 7 for *n*.

Teaching Note:
To verify students have understood this structure it is suggested that at least two more examples (of simply the loop structure) be discussed and reviewed during class time.

Two samples of source is provided here. It is suggested that one example as completed together on the board, and the 2nd, 3rd be attempted by the students under guidance from fellow students and from the teacher.

FOR *min%* = 25 TO 34 STEP 3
   PRINT "min is: "; *min%*
NEXT *min%*

FOR *tests* = 24 TO 0 STEP –2
   PRINT "Tests to go"; *tests*
NEXT *tests*

REPETITIONS – THE DO LOOP CONSTRUCT

The FOR loop structure fits well for when we know how often we need to repeat the program instructions. Many programming problems require repetitions which cannot be determined before the repetition is required. The DO LOOP is used when we **do not** know how often the loop will be repeated, but we do now which conditions we wish to be true to continue repeating the instructions.

The condition for repeating the loop is usually dependent on data being manipulated within the loop.

- DO NOT know how often the loop will be repeated
- DO KNOW conditions for repeating, conditions for stopping
- Condition uses data being manipulated in the loop.

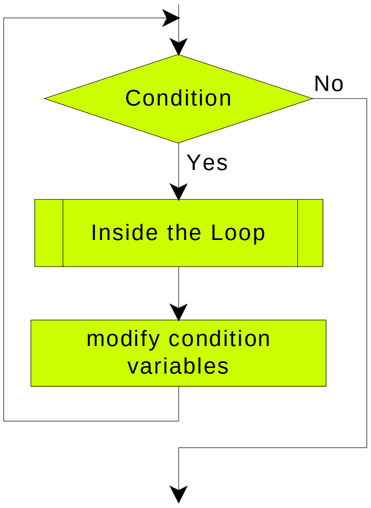
```
DO [{WHILE | UNTIL} condition]
     [statementblock]
LOOP
```

Entry-condition loop

```
DO
     [statementblock]
LOOP [{WHILE | UNTIL} condition]
```

Exit-condition loop

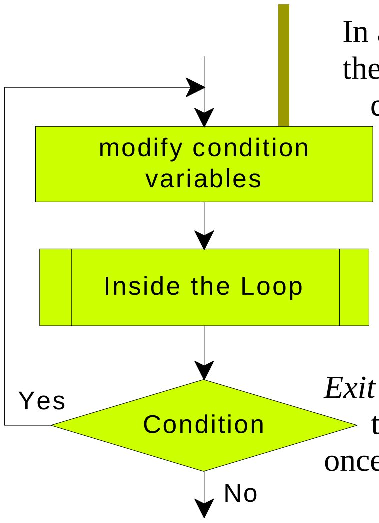
The DO LOOPs are control structures that let you repeat statements in a similar fashion to the for loop. The DO LOOPs are best used when you do not know how many times a loop should be processed (another term for it is *indeterminate loops*, or *conditional loops.)*



There are two forms for the DO LOOP structure; Testing the condition before performing the loop, or testing the condition after performing the loop.

In a *WHILE* clause, the looping continues as long as the condition is true. Looping stops when the condition becomes false.

In an *UNTIL* clause, the looping continues as long as the condition is false. Looping stops when the condition becomes true.



Entry Condition loops evaluate the condition **before** it executes the instructions inside the loop. Evaluating at the beginning (or top) means that if the condition is not met the instructions inside may never be executed.

Exit Condition loops evaluate the condition **after** it executes the instructions inside the loop. This means that the instructions in the loop will be executed at least once.

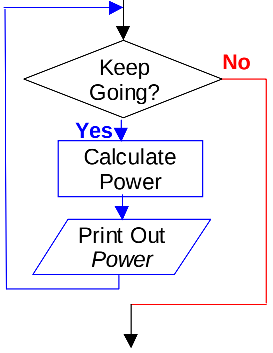
CASE STUDY. CALCULATING THE POWERS

We would like a program to list all the resulting powers of an integer that is less than 1,000 (the maximum resulting power).

For example:

$2^0=2$, $2^1=2$; $2^2=4$, $2^3=8$, $2^4=16$, $2^5=32$, $2^6=64$, $2^7=128$, $2^8=256$, $2^9=512$

Logic Review as Flow Chart



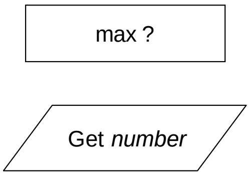
Determine the Purpose.

Stated above.

What are the Required Data.

- *n* the integer
- *maximum power* to keep the program from 'overflowing' we will set it to 1,000
- Power is calculated as n^i when i goes from 0 to infinite
- $n^0 = 1$ by definition of powers (indices)

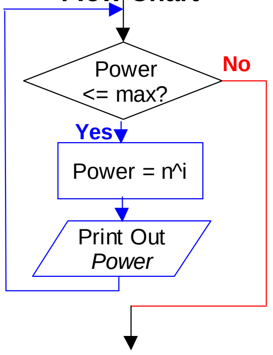
Required Data ?



Determine the Logic.

- Use variable *Power* to store the value of the calculated power n^i. $Power = n^i$
- Start with n^0=1, … and keep going up until n^m is bigger than the *maximum power*.
- We can mathematically calculate the Power by either using the formula $Power = n^i$ –or– use the relationship $Power = Power * n$
- We cannot use the FOR loop, because we do not know how many repetitions are required before we go into the loop.
- We do have a condition for exiting the loop (when *Power > maximum power*)
- We calculate the Power within the Loop

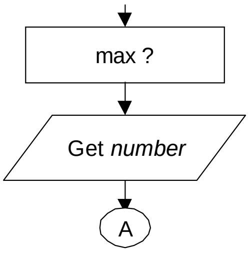
Updated Review as Flow Chart

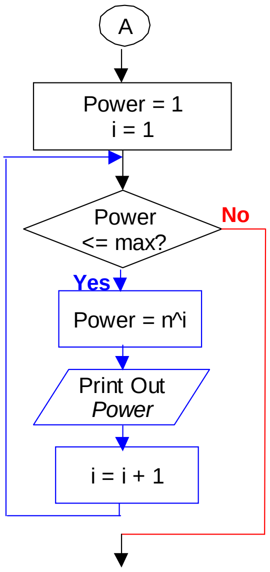


Draft the Computer Program.

- MaxPower is a constant and will hold the Maximum number we will approach.
- *n*, and *Power* are INTEGERS

Test & Re-test.





Listing

```
CONST MaxPower = 1000
DIM Power AS INTEGER
DIM n AS INTEGER

Power = 1  ' The first power n^0 is always 1

COLOR 10, 1
CLS
PRINT "This program prints all powers <"; MaxPower; "of an Integer."
INPUT "Enter an Integer"; n

DO WHILE Power < MaxPower
  PRINT Power;
  Power = Power * n
LOOP
```

Entry-condition loop

Learning Notes:

Demonstrate using n=2; the provided sample, and n=4 to let the students think through the mathematics and the looping condition.

Code Review; Desk-Check

If we were to test the program logic using the sample data (n=2) the DO LOOP structure works as discussed.

- When 1st we look at the Power < MaxPower;
- Power is currently 1 and MaxPower is 1000. The condition is true so enter the loop
  - Output the Power; 1
  - Calculate the next Power; Power = 1 * 2. Power is now 2
  - We hit the end of the loop, so we go back to the beginning
- When 2nd time we look at the Power < MaxPower. 2 < 1000 so the condition is true, enter the loop
  - Output the Power; 2
  - Calculate the next Power; Power = 2 * 2. Power is now 4
  - We hit the end of the loop, so we go back to the beginning
- When this time we look at the condition. 4 < 1000 is true, enter the loop.
  - Output the Power; 4
  - Calculate the next Power; Power = 4 * 2. Power is now 8
  - We hit the end of the loop, so we go back to the beginning
- We continue doing this until the condition is no longer true. When the condition is false, the program continues with the next statement after the loop.

Group Exercise

1. Draw the completed flow-chart for the sample program.

In Class Exercise

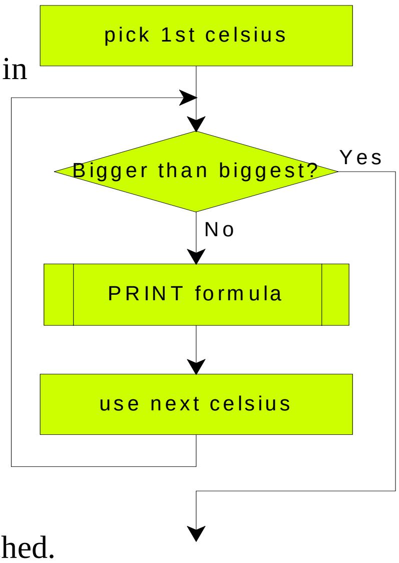
1. Write down what the program output will be if the user enters 5 for *n*.

2. Write down what the program output will be if the user enters 10 for *n*.

3. Write down what the program output will be if the user enters 20 for *n*.

CASE STUDY. A CONVERSION TABLE

Write a program to print out a table of temperatures in Celsius and Fahrenheit.



Determine the Purpose.

Stated above.

What are the Required Data.

- Beginning and ending value for the calculations (smallest Celsius to Largest Celsius)
- Use the mathematical formula *F = 1.8C + 32*

Determine the Logic.

- Start with the smallest Celsius
- Calculate F (Fahrenheit)
- Print both the Celsius and Fahrenheit to the screen
- Repeat the above until biggest Celsius number is reached.

Draft the Computer Program.

- minCel, maxCel to hold the beginning and ending number
- *celStep* to hold how many Celsius up we move before calculating the next Fahrenheit.
- Celsius we will increment in whole numbers, so we use the INTEGER type
- Fahrenheit uses a 1.8 calculation so we need to use a SINGLE or DOUBLE. *(ie. decimal values needs to be stored.)*

Test & Re-test.

Code Review; Desk-Check

- When 1st we look at the *Celsius <= maxCel*;

```
REM
' Prints a table of Celcius and Fahrenheit equivalents.

CONST minCel = -10   ' minimum Celsius temperature
CONST maxCel = 30    ' maximum Celsius temperature
CONST celStep = 5    ' increment between Celsius values

DIM Celsius AS INTEGER
DIM Fahren    AS SINGLE

COLOR 14, 7
CLS
PRINT "This program prints a table of Celsius and Fahrenheit temperatures"

Celsius = minCel

PRINT "Celsius  Fahren"
DO WHILE Celsius <= maxCel
   Fahren = (1.8 * Celsius) + 32
   PRINT USING "###     ###.#"; Celsius; Fahren
   Celsius = Celsius + celStep
LOOP
```

- *Celsius* is –10 and *maxCel* is 30. The condition is true so enter the loop
  - Calculate *Fahren*. Fahren is now 14
  - Print the value of Celsius and Fahren to the screen
  - Calculate *Celsius* = *Celsius* + *celStep*; *Celsius* is now –5
  - We hit the end of the loop, so we go back to the beginning

- When 2nd time we look at the *Celsius* <= *maxCel*; *Celsius* is –5 and *maxCel* is 30. The condition is true so enter the loop
    - Calculate *Fahren*. Fahren is now 23
    - Print the value of Celsius and Fahren to the screen
    - Calculate *Celsius* = *Celsius* + *celStep*; *Celsius* is now 0
    - We hit the end of the loop, so we go back to the beginning
- When this time we look at the test condition; 0 < 30 is true so enter the loop
    - Calculate *Fahren*. Fahren is now 32
    - Print the value of Celsius and Fahren to the screen
    - Calculate *Celsius* = *Celsius* + *celStep*; *Celsius* is now 5
    - We hit the end of the loop, so we go back to the beginning
- We continue doing this until the condition is no longer true. When the condition is false, the program continues with the next statement after the loop.

Group Exercise

1. Draw a more detailed flow-chart of the above program.

In Class Exercise

1. Write down what the program output will be if celStep were 10 instead of 5.

2. Write down what the program output will be if celStep is 5 and maxCel is 10.

CASE STUDY. SETTING A GOAL AND WORKING TOWARDS IT

We are interested in finding out whether, and how much money we need to save in our Bank Account to retire.

Determine the Purpose.

Stated above.

What are the Required Data.

- Target Savings
- Deposit Amount
- Bank Interest Rate
- Length of Deposit

Determine the Logic.

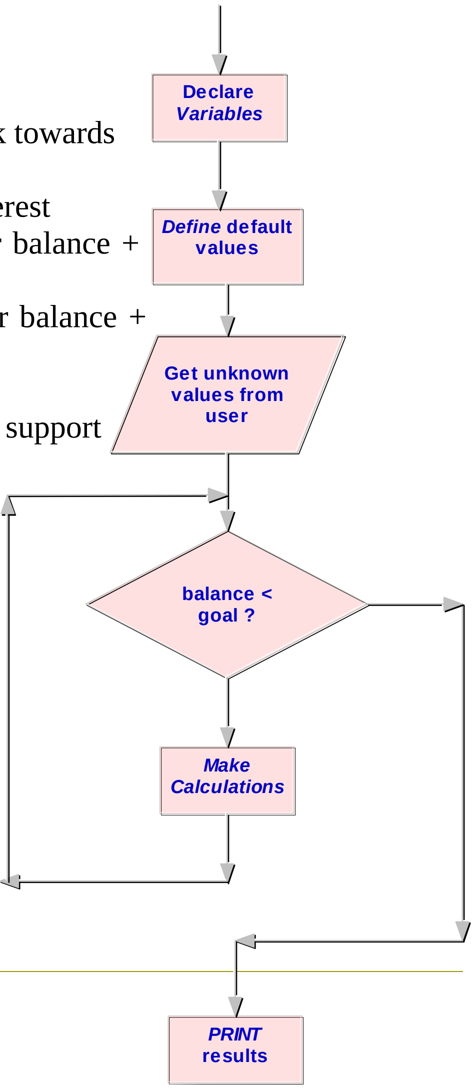
- Start with a balance of zero, and a goal to work towards
- A deposit amount is taken
- The balance after the first year is deposit + interest
- The balance after the 2nd year is the 1st year balance + interest
- The balance after the 3rd year is the 2nd year balance + interest

Draft the Computer Program.

- *goal* for the retirement amount goal, must support coins as well as dollars.
- *payment* for the initial deposit amount, must support coins as well as dollars.
- *balance* for the current balance in the account, must support coins as well as dollars.
- *interest* for the interest rate being charged to the account, must support percentages (decimal values.)
- Symbolic Representation:
  - new balance = current balance + interest
  - balance = balance * (1 + interest rate)

Test & Re-test.

Code Review; Desk-Check



```
DIM goal AS DOUBLE
DIM interest AS DOUBLE
DIM payment AS DOUBLE
DIM years AS INTEGER
DIM balance AS DOUBLE
balance = 0

PRINT "So you want to put some money aside for retirement, hmmmm."
INPUT "How much money do you need to retire"; goal
INPUT "How much money will you contribute every year"; payment
INPUT "Interest rate in % (eg. use 7.5 for 7.5%)"; interest

interest = interest / 100
balance = payment
DO WHILE balance < goal
   balance = balance * (1 + interest)
   years = years + 1
LOOP
PRINT
PRINT "You can retire in "; years; " years"
PRINT "with "; balance; "in the bank"
```

Group Exercise

In Class Exercise

1. Using the previous Code Review; Desk-Checks, perform your own Code Review on how the program will process the program using the following sample data:

goal = 50,000
interest = 5.0%
payment = 5,000

Commentary on Loop Structures?

Kernighan and Ritchie (*the designers of the technical programming language called "C"*) estimate that the exit-condition (evaluating after the loop) is needed for about 5% of loops.

There are several reasons why computer scientists consider an entry condition loop superior. One is the general principle that it is better to look before you leap (or loop) than after. A second point is that a program is easier to read if the loop test is found at the beginning of the loop.

Finally, in many uses, it is important that the loop be skipped entirely if the test is not initially met.

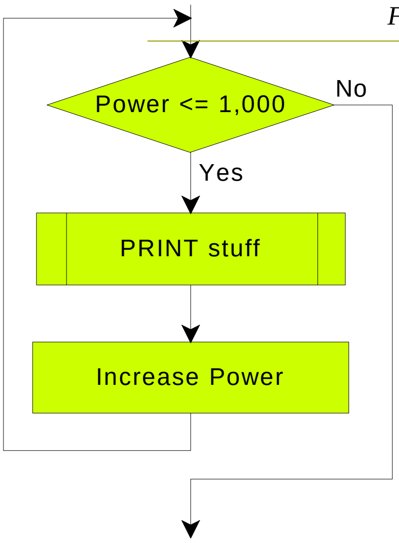
Repeating – The Exit Condition Loop

After each execution of the loop-body, the exit-condition is evaluated. If the exit-condition is true, loop exit occurs and the next program statement is executed. If the termination exit-condition is false, the loop-body is repeated.

```
DO WHILE Power <= MaxPower
  PRINT Power;
  Power = Power * n
LOOP
```

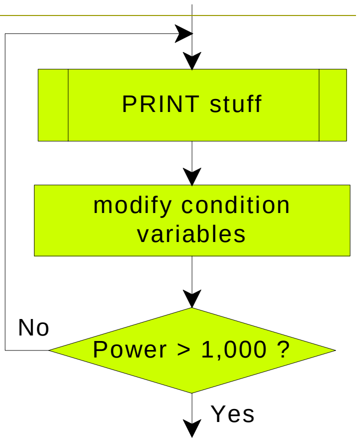
```
DO
  PRINT Power;
  Power = Power * n
LOOP UNTIL Power > MaxPower
```

Figure: Entry condition (on left) and exit-condition (right) statements.



By rearranging the logic used in our earlier *entry-condition* loop we now have an *exit-condition* loop performing the same calculation. This will only work properly if the user had entered an n > 0.

The exit condition loop checks the condition for repeating the loop (iteration) after the execution of the loop.



```
nguesses = 1
PRINT "Guess #:"; nguesses;
INPUT guess
DO WHILE guess <> numb AND guess <> 0
 nguesses = nguesses + 1
 PRINT "Guess #:"; nguesses;
 INPUT guess
LOOP
```

```
nguesses = 0


DO
 nguesses = nguesses + 1
 PRINT "Guess #:"; nguesses;
 INPUT guess
LOOP UNTIL guess = numb OR guess = 0
```

Figure: Entry condition (on left) and exit-condition (right) statements.

There are very legitimate times when an *exit-condition* loop makes more sense than rewriting your logic (algorithm) to fit the *entry-condition* loop. The above exit-condition loop requires fewer instructions than the entry condition loop.

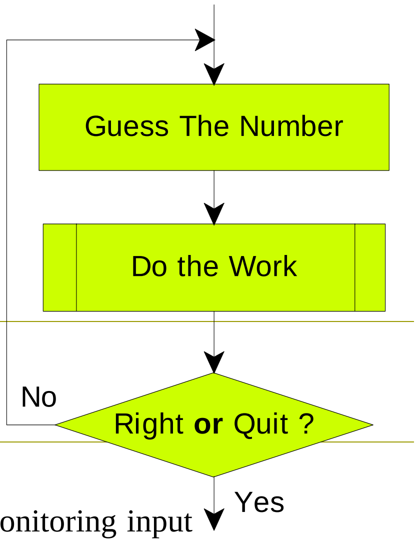
```
PRINT "Press Esc to exit...";
'{stuff to be done before checking the
“Esc” has been pressed}
DO WHILE INKEY$ <> CHR$(27)
' {stuff to do}
LOOP
```

```
PRINT "Press Esc to exit...";


DO
  ' {stuff to do}
LOOP UNTIL INKEY$ = CHR$(27)
```

Figure: Entry condition (on left) and exit-condition (right) statements.

The entry-condition loop shown above indicates that for some algorithms an entry-condition loop can get complicated trying to duplicate code better implemented as an exit-condition loop.

Keep the use of exit-condition loops to cases which require at least one repetition of the logic/algorithm. For instance, the logic for a number guessing game is a good fit for using the exit-condition loop.



```
DO
'  player guesses the number
LOOP UNTIL {player quits -or- guess is correct}
```

A common use of the exit-condition loop is for monitoring input where a specific value is desired.

```
COLOR 10, 1
CLS
LOCATE 25, 1
PRINT "Press Yes or No to continue (Y/N)"

DO
 A$ = INKEY$
 LOCATE 12, 25
 PRINT "You entered "; A$

LOOP UNTIL A$ = "Y" OR A$ = "N"
```

LOCATE moves the cursor to a specified position on the screen.

LOCATE [row%] [,column%]

■ row% and column%   The number of the row and column to which the cursor moves.

Code Review; Desk-Check

Inside the loop

- A$ is given a character from the keyboard.
- The cursor is put at row 12; column 25
- Output the entered character
- At the end of the loop check whether "Y" or "N" has been entered. If condition true, continue at the end of the loop, otherwise repeat the loop.

INKEY$ Reads a character from the keyboard.

- INKEY$ returns a null string if there is no character to return.
- For standard keys, INKEY$ returns a 1-byte string containing the character read.
- For extended keys, INKEY$ returns a 2-byte string made up of the null character (ASCII 0) and the keyboard scan code.

Group Exercise

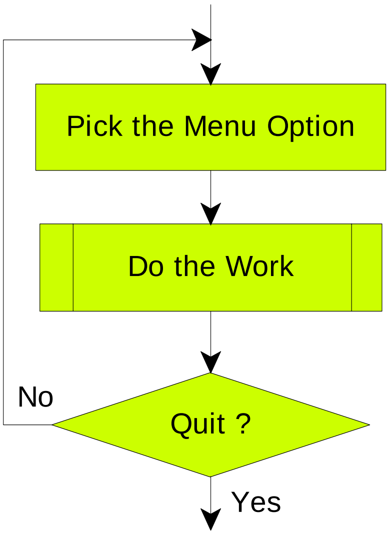
1. Draw a flow-chart of the program flow in the code listed above.

Exit-Condition Menus

An example of where exit-condition loops are more appropriate than entry-condition loops are programs with “menu” choices from which the program user selects a program operation. The menu for a statistics program might look as follows.

```
1. Compute an average
2. Compute a standard deviation
3. Find the median
4. Find the smallest and largest value
5. Plot the data
```

```
0. Quit / Return to Previous Menu
Enter your choice (1 through 5):
```



An exit-condition loop, can repeat, let the user keep using the above menus until the user chooses to quit by selecting 0. One thinking algorithm for the above menu would be represented in the below code design.

```
do

  { whatever the work is to be done within the menu
         selection }

loop until nchoice = 0
```

Putting together a sample of just the above menu, without the actual work that is to be done with the user selection would look something similar to the following sample code.

Group Exercise

1. Draw a flow-chart of the above code program flow.

In Class Exercise

1. Using the previous Code Reviews as an example, describe what the program above will do when the user enters the following data samples:

SAMPLE CODE: MENU SYSTEM

```
CLS
  PRINT "Which statistical Analysis do you wish"
  PRINT
  PRINT "   1. Compute an average"
  PRINT "   2. Compute a standard deviation"
  PRINT "   3. Find the median"
  PRINT "   4. Find the smallest and largest"
  PRINT "   5. Plot the data"
  PRINT
  PRINT "   0. Quit / Return to Previous Menu"

DO
  INPUT "Enter your choice (0 through 5) "; a%
     ' Use SELECT CASE or IF to check the user
     ' input, and do the work that has been requested

LOOP UNTIL a% = 0
```

1, 3, 4, 2, 0, 5, 3

COMBINING THUS FAR

The following program code brings together a number of the flow control structures we have reviewed thus far and shows the utility (usefulness) of the exit-condition loop.

Be warned: The program uses a number of the programming design structures discussed earlier.

Problem

Everyone likes playing games, and we want to write a game using the old guessing technique, but this time the computer will pick the number and our work will be to try and guess the number the computer has picked. The fewer the guesses we make the better we are at playing the game.

Determine the Purpose.

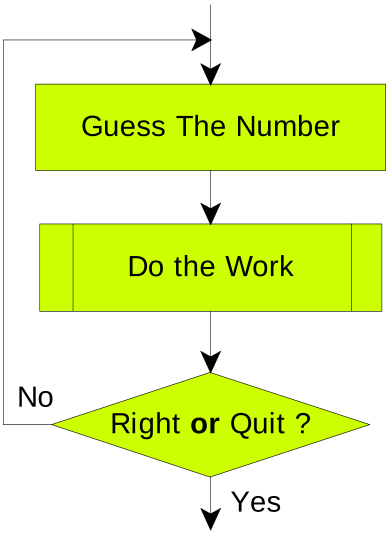
Stated above.

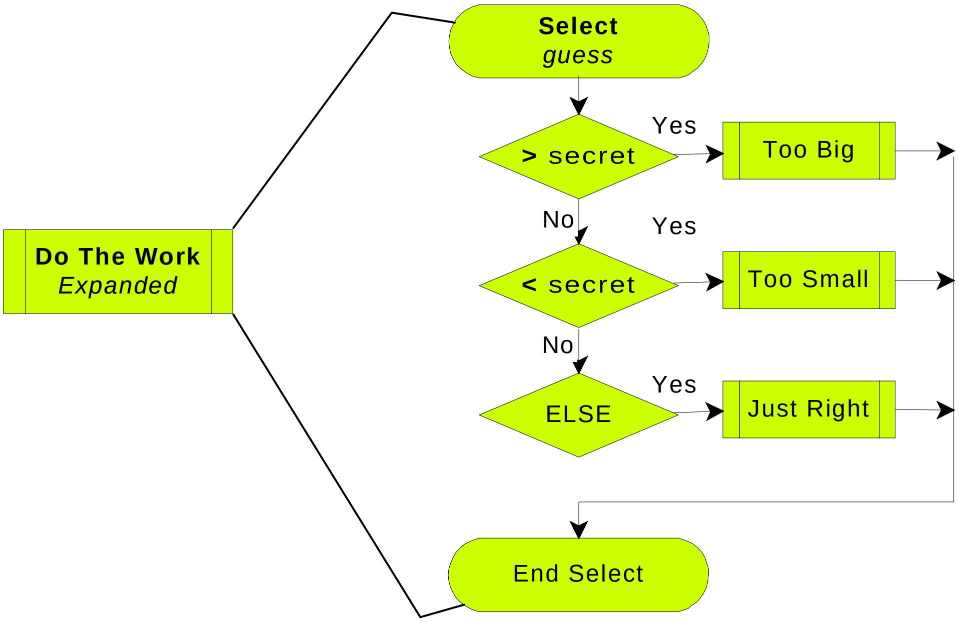
What are the Required Data.

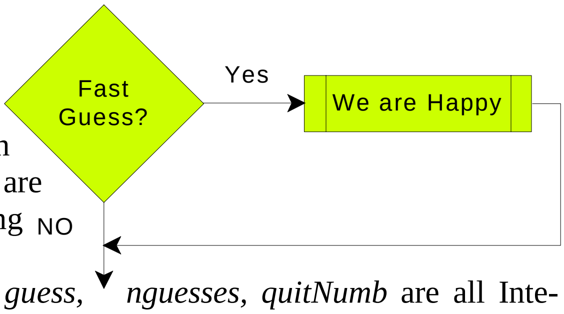
- We need a secret number for guessing
- We need the users guess
- To limit the size of the game we want to limit it somehow
    - We need the biggest number allowed
    - We need the smallest number allowed

Determine the Logic.

- Our program picks a secret number
- The user takes a guess, and we keep track of how many guesses are being taken.
- We check whether the guess got it right.
    - If they guessed the number, the game is over.
    - If they guessed a number lower than our secret number, we tell them it is too low, go back and let the user make another guess.
    - If they guessed a number higher than our secret number, we tell them it is too high, go back and let the user make another guess.
- For the really fast guesses, we give the user a



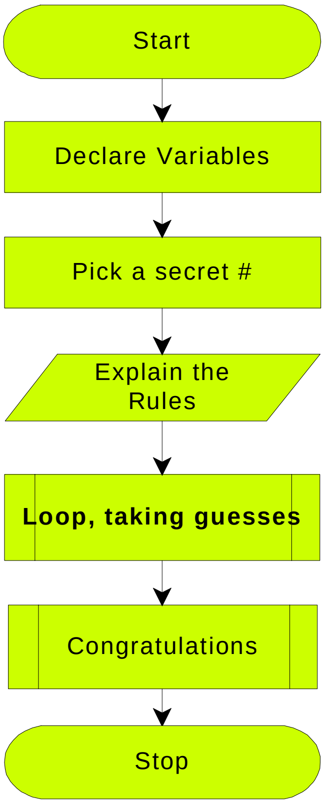




little extra congratulations.

Draft the Computer Program.

- *maxNumb* and *minNumb* will contain our largest and smallest number, and are constants that do not change during the program.
- *numb* is the number to be guessed, *guess*, *nguesses*, *quitNumb* are all Integers and hopefully self-explanatory variables (put additional info in source code)
- Picking a secret number in our game is the same as picking a 'random' number. For QBasic this is used through the below functions documented in the QBasic HELP screens.
  - RANDOMIZE TIMER
  - numb = INT (RND * maxNumb) + minNumb
- Tell the player the rules of the game, and we have picked a secret number to be guessed.
- Repetitions. We do not know how many guesses it will take the user to win the game, so we will use a DO LOOP.
  - We use an exit-condition loop, because the user must tell us whether they wish to continue, or not play the game.
  - Record the number of guesses performed with each loop.
  - Checking the Guesses. We can use either the IF or the SELECT CASE. The SELECT CASE looks cleaner, so we will use that. (logic stated above)
- Checking whether further congratulations should be given. Since we used the SELECT CASE previously, we will use IF here.



Test & Re-test.

RANDOMIZE initializes the random-number generator.
TIMER Returns the number of seconds elapsed since midnight.
RND returns a single-precision random number between 0 and 1.

RANDOMIZE [*seed%*]
RND[(*n#*)]

seed% A number used to initialize the random-number generator. If omitted, RANDOMIZE prompts for it.

n# A value that sets how RND generates the next random number:

n#	**RND returns**
Less than 0	The same number for any n#
Greater than 0 (or omitted)	The next random number
0	The last number generated

Example:

```
   RANDOMIZE TIMER
   x% = INT(RND * 6) + 1
   y% = INT(RND * 6) + 1
   PRINT "Roll of two dice: die 1 ="; x%; "and die 2 ="; y%
```

THE NUMBER GUESSING GAME

REVIEW EXERCISES.

1. Draw a detailed flow-chart diagram of the above exit-condition program.
Hint: Combine the separate flow-charts we have reviewed for each of the separate items.

2. The Number Guessing game limits the guessing between 1 and 100. Which variables will you modify to let the program to ask the user to provide the smallest number and the biggest number.

3. The Number Guessing game does not provide a way for the user to continue playing the game, without having to restart the program. Draw a flow-chart indicating the changes you would make to allow the user to select to play another game.
Hint: Finish Question 1, and this part is much easier.

♦

Sample Source Code Solution

```
' Program:
'
' Author:
' Class:
'
' Purpose:
' This is a guessing game program to highlight the use of flow control
' where the user tries to guess a secret number
'
' (a) Start
' Declare variables
CONST maxNumb = 100      ' The biggest number
CONST minNumb = 1        ' The smallest number

DIM guess AS INTEGER     ' The guess
DIM numb AS INTEGER      ' The secret number
DIM nguesses AS INTEGER  ' The number of guesses
DIM quitNumb AS INTEGER  ' The number the user can use to Quit

' Generate a random number for the user to guess
' Put the secret number into ‘numb’
RANDOMIZE TIMER
numb = INT(RND * maxNumb) + minNumb
quitNumb = minNumb – 1
' Explain the game
'
CLS
PRINT "Number Guessing Game : Charley Alpha"
PRINT "I have picked a secret number between and including";
PRINT minNumb; "and"; maxNumb
PRINT "How many times will it take you to guess the number?"
PRINT "If you want to give-up at any time, just enter"; quitNumb

nguesses = 0
DO
    nguesses = nguesses + 1
    PRINT "Guess #:"; nguesses;
    INPUT guess
    SELECT CASE guess
    CASE IS > numb
        PRINT "Too high"
    CASE IS < numb
        PRINT "Too Low"
    CASE ELSE
        ' They got it so we should be going out soon
        PRINT "hmmmm"
    END SELECT
LOOP UNTIL guess = numb OR guess = quitNumb

PRINT
IF guess = numb THEN
    PRINT "Congratulations"
    IF nguesses < 10 THEN
        PRINT nguesses; "guesses is real fast smart one. ";
    END IF
    PRINT "You got it"
ELSE
    PRINT "Quitter !"
END IF
'  STOP
```

Declare the variables to be used through the program.

Pick a secret number

Explain the game so the person using it understands

Loop, waiting for the user to guess the correct number, or get it wrong. While checking each guess we tell the user whether they are guessing higher or lower.

A little incentive for the user to play again, tell them how good they are if they guess correctly and fast.

Module Review Questions

1. A very common term in programming is “VARIABLE”. Define what a variable is:

2. There are two types of variables. List, describe and give an example for each of these two types

3. There are many ways of assigning data to a variable. Using an example, describe the ways of storing a value in a variable

4. Comment lines are very common in programs. Give a reason why comment lines are included in programs

5. There are two ways of writing comments on a program. Identify each method

6. For each of the following reserved words, Describe its purpose and write down a simple example

   a) PRINT
   b) DATA
   c) CLS
   d) END

Sources and References

Butkus, Chuck, teach yourself QBasic, 2nd Ed., (New York, MIS Press, 1994)
Cornell, Gary and Cay S. Horstman, Core Java, (Mountainview, SunSoft Press, 1996)
Hergert, Douglas, QBasic Programming for Dummies, (Foster City, IDGBooks, 1994)
Koffman, Elliot B. Problem Solving and Structured Programming in Modula-2, (Reading, Addison Wesley Publishing Co., 1988)
Liberty, Jesse, Teach Yourself C++ in 21 Days, (Indiana, SAMS publishing, 1994)
Microsoft, QBasic Help, (Redmond, Microsoft Corporation, 1995)
Presley, Bruce and Freitas, William, A Guide to Structured Programming in BASIC for the IBM PC and Compatibles 3rd Ed. (Pennington, Lawrenceville Press, 1992)
Salmon, Steven. The Beginners Basic Helpfile v2, (Hampshire, na, 1997)
Waite, Mitchell & Stephen Prata, Donald Martin, C Primer Plus (Indianapolis, Howard W. Sams & Co., 1984)

Notes:
Source code for examples in these exercises should be available on the school server for student access.

Sample solutions/ outputs for the projects are provided for students in executable form only.

http://www.tongatapu.net.to/compstud/ - Computer Studies Course Notes
http://www.tongatapu.net.to - **Tonga** on the **'NET**

http://www.tongatapu.net.to is available on all networked computers at Queen Salote College and participating Schools.

Wednesday, February 09, 2000

To Do List – So it stares at me until it is done

- Functions and Subroutines
- File Input/Output
- String Manipulation
- Data Structures