QB Now!

BY NEO DEUS EX MACHINA
HTTP://WWW.HARSOFT.TK

PART I: THE BASICS

INTRODUCTION

Welcome to the QB Now! Tutorial, part one, explaining the basic things you need to know of and about QB. First of all, let me introduce myself. I am Neo Deus Ex Machina, but usually Neo to keep things easy. I started programming quite some years ago, I was 7 years old when I first saw a computer… with QB on it! I followed simple courses with the local programming club and started programming myself. After 6 years of programming, I didn't know much more than 6 years before. But when I got to high school, where there was (free for me) internet, I started learning, faster and faster. The last 4 years up till now my programming skills were increasing rapidly. According to some people I may even be called a QB Expert, and I'm proud of that reputation, after 10 years of programming so far. But there are still better people who changed the whole QB world with their programming and I hope that one day I'll change the QB World as well. But that's for later.

The main objective I'm trying to accomplish by writing these tutorials is that I can copy some of my knowledge into the reader's. I.e. it strives for the keeping of knowledge, and for preventing QB to die soon. The more people know much about QB Programming the better. Don't let QB go extinct.

If you have any questions, remarks or other messages about this tutorial or anything found in it, please let me know. My email address is arecinos@gmx.net. I hope you like reading this tutorial and will be somewhat wiser after finished reading it! Enjoy!

CONVENTIONS

In this tutorial some layout conventions will be used to clarify and to make the tutorial much more readable.

Code sections will be written as follows:

```
CODE SECTION (Courier New Bold)
```

Remarks or notes are given as follows:

Remark or note section (TNR italic blue)

The following are warnings or exclamations:

Warning or exclamation section (Arial bold red)

That was all so far, I think.

Let's start with the real tutorial now.

CHAPTER I: GETTING QB

GET THE QB IDE

First of all, to program in QB, you need to have a QB compiler or interpreter. You can get several from the internet. I advice you to download QB 4.5, the best version of QB, together with PDS. QB45 is covered in these tutorials. You can get QB45 from *http://www.qbnz.com* for instance.

If you have downloaded the package, extract it into a folder you like (e.g. C:\QB). Now go to that folder and run *QB.EXE*, which is the QB IDE. You'll now see a blue screen, with a menu bar on the top. Go to options and make sure *Full Menus* is checked. This should enable all features of the QB IDE.

Now, that blue box on the screen is the programming area, in which you can type your code. In the *File* menu there is *Save* and *Open* possibilities, used for saving to and loading from harddisk respectively. There is also an *Exit* if you want to quit. If you're used to Windows or a similar Operating System, you should have no trouble finding the means of the menu commands and how to use them. Next we'll discuss the general basics about programming.

If Windows gives you an warning message about the working directory of QB.EXE being invalid, right-click on QB.EXE, go to properties and program. In the two fields Command Line and Working Dir type in the correct path of your QB.EXE version.

MAKING AN EXE IN QB

Before we start with the actual programming, I'll first discuss the possibility of QB to generate EXE files from your code. These executables are somewhat faster than running in the QB IDE. Furthermore, if you distribute your program, most people would like EXE files better than the complete source code.

In the menu, select *Run*. There must be an option *Make EXE*. If you click that option, you'll get to the make exe screen. Here you can select if you want a standalone or a dependent EXE. I advice you to always select *Standalone* because the other requires certain libraries (*BRUN45.LIB*) to be on the computer of the person who runs the program, and not all people have this library. Selecting standalone will incorporate this library into your exe, so that anyone can run it, as the library is also in the exe file. Your exe file will be a bit bigger, but that is of no importance (as a couple of kB's extra isn't much considering all those almost-terabyte harddisks nowadays). So select *Standalone*, always. Next you can choose if you want to *Produce Debug Code*. This should be off. Then you can type the name of your exe file in the textbox, but usually it's already typed in the box. Then select *Make EXE* and the exe file will be made! Voila.

CHAPTER II: PROGRAMMING BASICS

Programming is not just a thing you do as if you were eating. Programming involves knowledge, true. But the main thing it involves is thinking, and thinking like a computer would do. I can't explain anyone *how* to think, so the hardest part of learning to program is learning to think. You'll have to learn it yourself as no one can teach you to. Now I'll discuss the general programming idea that every high programming language uses. It is also immediately a hard part for real beginners, so I'll try to explain it as well as I can.

VARIABLES

Variables are parts of the memory your program uses for storage of data. This data can either be numbers or text. Variables can store different types of data, depending on the data type the variable is of. The different types of data are discussed after this part. E.g. variables can be of type *number*, which means your program can store numbers in that variable. Other variables can be of type *string*, which indicates that you can store text in that variable. Variables are very handy things because you can also do something with them, e.g. you can add to *number* variables together to calculate the sum, or you could append two *string* variables to eachother. In other words, you can perform tasks with them, that's why they're there. Your program can retrieve the contents (*value* later on) of a variable, as well as putting something in the variable (*assigning* later on) or perform calculations with it. The best way to get used to variables is using them and seeing them be used. That's why I'll continue now.

If you don't understand it yet, consider comparing variables with drawers of your desk. You have small and large drawers, in which respectively small and large stuff can be put. As already implied, you can put stuff in the drawers, you can also see what's in the drawer. You can also remove stuff from the drawer. I learned this theory about variables using this simple example.

DATA TYPES IN QB

There are six different indigenous datatypes in QB. I'll now discuss them one by one.

Integer – the most common datatype. It can only hold numbers without decimals ranging from –32768 to 32767. This datatype is 2 bytes in size.

Long – mostly used after integer, it can hold larger numbers than an integer. Viz. ranging from –2147483648 to 2147483647 without decimals. The long is 4 bytes in size.

Single – A datatype which is also able to hold numbers, but with decimals. It can hold numbers from ±1.401298^{E}-45 to ±3.402823^{E}+38, with a precision of 7 digits. The single is 4 bytes in size.

Double – A datatype bigger than a single, it can hold decimal numbers ranging from ±4.940656^{E}-324 to ±1.7976931^{E}+308, with a precision of 15 digits. The double is 8 bytes in size.

String – This is a datatype that can hold text or characters. A maximum of 32767 characters is the string's limit. Strings are noted with double apostrophes (") around it.

String * n – This is the same as string, only that it has a fixed size of n characters.

That were all data types for QB. There is a way to create your own (pseudo) data type, but that's discussed in on of the next parts of QB Now!. You can declare variables to be one of the datatypes implicitly or explicitly. Explicit means that you type a statement explicitly saying that the variable is of that datatype. Implicitly means you just use the variable with a post-fix:

% - Integer
& - Long
! – Single
- Double
$ - String

So a variable called MyVariable% is always an integer, even if you didn't explicitly tell QB. About explicit declaring, it follows later.

REMARKS OR COMMENTS IN QB

When programming a lot of code, you may forget or have forgotten what some parts of your program do. In order to make it easier to find out again what the piece of code was meant for, there is the ability to add remarks to the code in QB. Remarks or comments are written as if it were code, but they are not executed by QB. You can write anything in remarks, QB won't even look at it ☺. This way, you can just put in remarks during your code what happens at the very moment in your code.

In QB, you can add remarks on two ways. The first, and mostly used, way is using the apostrophe ('). Everything after the apostrophe is a remark, and QB won't look at it. The second way is using the command *REM*. Everything after this command is remark. In this tutorial, I'll use the apostrophe-way, for most programmers do it the very way.

COMMANDS: STATEMENTS AND FUNCTIONS?

In QB you can give commands in order to do something. As you will learn soon, there are commands for printing text on the screen, or changing colors. However, commands can be divided into two groups, which are important. It's also important you know this.

First of all there are the statements. These are just the regular commands doing something without returning something. Examples of statements are the text-printing command and the color switching command.

Then there are the functions. Functions are commands that not only do something, but also return something. Usually commands for retrieving data are functions, because they return the data you requested. E.g. there is a command for checking which character is on the screen at a certain location, and this function returns the character found at the location you specified.

If you understand it now already, you're brilliant. Usually you'll understand it when you see it working. Now, let's start with programming!

CHAPTER III: FIRST PROGRAMMING

OUTPUT OF TEXT

Here, this is your first program:

```
PRINT “Hello! This is my first program!”
END
```

Type this into the code window of the QB IDE. Then run the code…

To run your code press [Shift] + F5.

What did you see? ☺ By looking what a program does when you know what the code is, is a better learning method than just reading about what it is supposed to do, but it is a convention and I won’t do else.

PRINT [var]	This command (statement) prints text on the screen. It accepts any variable or data. If you don’t specify data to print, it prints an empty line.
END	Ends the program immediately.

This should be clear for all. ☺ But there’s more you can do with text on the screen. For example think of colouring…

```
COLOR 4
PRINT “This is red”
COLOR 1
PRINT “How about blue?”
COLOR 5, 7
PRINT “Purple on grey”
END
```

Using the Color statement you can define the current color for printing text on the screen. Here is a table holding the color values and its attached color.

0	Black	8	Dark Grey
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	Grey	15	White

You should know these colors by heart. Note that a color and it’s lighter version always differ 8, e.g. cyan = 3 and light cyan = 3 + 8 = 11.

The color statement is able to set the current foreground and background color used for the PRINT command. Only colors 0-7 can be used as background.

COLOR fground% [, bground%]	This command sets the foreground and/or background color to be used with PRINT commands. All following PRINT commands adapt these colors until another color statement.

There's still yet some other things that can be done with text on a screen. The following example tells you how to locate and clear text.

```
CLS
LOCATE 12, 24
PRINT "Text in the middle of the screen"
LOCATE 25, 1
PRINT "Bottom";
END
```

As you can see when you run this program, the text will be placed on various positions on the screen, as specified in the code. Now, how does this locate work? The screen is divided into blocks. In each block fits one character. Normally, the screen his 80 blocks horizontal and 25 vertical, which means you can print 80 characters horizontally and 25 lines on the screen. To specify a place on the screen, you should use coordinates, which is a set of numbers (in this case 2) which are the vertical (y) and horizontal (x) position on the screen. The upperleft corner has coordinates 1, 1 and the lowerright 25, 80. Locate first requires the y-coordinate (vertical) and then the x-coordinate (horizontal). The next print statement will then be executed there.

Note that printing to line 25 in the example required a semicolon after the print. This is due to a simple, though stupid, reason. When you would print on lines 24 or 25 normally, the screen would shift up, so that the lines on top of the screen disappear. To prevent this shifting up, a semicolon should be added after the print statements which print on any of the lines 24 or 25.

`LOCATE y%, x%`	Sets the coordinates on the screen for the next print command. Everything sent to the next print command will be printed on that position.
`CLS`	This command clears the whole screen. All text and graphics printed on screen will disappear.

FIRST TIMING METHODS

In this paragraph you will learn how to time you programs a bit. Take a look at the following example:

```
CLS
PRINT "Please wait 3 seconds or press a key"
SLEEP 3
PRINT "Ok, now press any key to quit"
SLEEP
END
```

As you can see when you ran this program, the sleep command can do two things. When you specify a number after sleep, it waits for the specified number of seconds, or for you to press a key. Sleep with no parameters (arguments, data passed to the command) waits until the user presses a key.

`SLEEP [x]`	Wait for x seconds or wait until a key press.

As already told, sleeps can be skipped by pressing a key. However, when you press a 'normal' key, the keybuffer will be filled and you computer may beep, and your program will *hang for a short while. To prevent this, you can also skip sleeps with the Control key without filling the keyboard buffer.*

USING VARIABLES

Finally time for more dynamic stuff. The variable part in the general section wasn't there for nothing. Now we finally get to use them, and you can learn how they work. Take a look at the following example.

```
CLS
DIM MyName AS STRING
DIM MyAge AS INTEGER
MyName = “Neo”
MyAge = 17
PRINT “Hello I am ”; MyName
PRINT “and I am ”; MyAge; “ years old”
END
```

Ok, there is some new stuff in this example, which will be explained soon. As you could see when you ran this example, the value of both variables MyName and MyAge was print on the screen on their respective locations specified in the print statement. Variables are means to store data (temporarily) for later use. In this case, the data is used with the print statement. You can assign a value to a variable using the equals (=). Also make sure that the variable you assign data to is able to hold such data. E.g. don't try to assign text to a numerical variable.

Now lets discuss the making of a variable. The way used in the above example is the *Explicit* way of declaring a variable. This means, you explicitly tell the QB IDE you want a variable of that type with that name, using the command *DIM* (which stand for *Dimension* which is the same as creating a variable). Then further in the program, you can use your variable by just typing its name.

Secondly, there's something new with print. In the example some semicolons are placed. With print, you can use either commas (,) or semicolons (;) to add different texts with variables together. If the variable is a string, you can use the + operator to add text and the variable together. Compare the output of the follow program:

```
MyName$ = “Neo”
PRINT “My name is: “ + MyName$
PRINT “My name is: MyName$”
```

When you compare the output of the first and second print, you'll notice the difference. If you want to use a variable in your text, you should separately add it to the text already there, in this case + has been used (only for strings to text), but a semicolon or comma can also be used.

The second print in this example will actually work the same as the first in PHP.

Note that in this example the *Implicit* way of declaring a variable has been used. As you can see, there is no command explicitly declaring the variable of a certain type. With implicit declaring, you just use a variable as if you declared it, then it's up to QB to see that a new variable has been created. Implicit declaring requires postfix type declarations, so that QB can still see what type the variable is of. The variable MyName in the example is of type string ($). If you don't specify a postfix, the standard datatype will be used (usually Single). There is a way to change the standard datatype, but that's for later. ☺

Keep an eye on where to put quotes. Print MyName$ is different from Print "MyName$". The first prints the value of MyName$ on screen, the second prints MyName$ literally to screen.

`DIM varname [AS vartype] [, more]`	Explicitly declares as variable with the name varname as vartype. If the latter part is omitted, varname will have the standard datatype. You can declare multiple variables using on line of dim.

`PRINT [var] [,] [var..]`
`PRINT [var] [;] [var..]`
	Prints a sequence of data to screen. Commas or semicolons are used to append certain data for screen output. If you use a semicolon, the two variables will be printed together without spaces between them. Using the comma however causes both variables to be print on screen with a certain amount of spaces between them. Print is not limited to printing two variables, you can specify much more. Also, if you omit the last variable while still placing a comma or semicolon at the end, the next print command will start on the very place.

To see what's meant with the last sentence of the print-explanation (about omitting the last variable while still placing a comma or a semicolon), I give you an example:

```
PRINT "Hello ";
PRINT "You.",
PRINT "How are you?"
```

When you ran this, you saw that you got attached to hello and the last question was still on the first line after you, but with a couple of spaces between it. Try experimenting with this as much as needed to understand it fully. ☺

ASKING THE USER FOR INPUT

As you might have noticed, this is not very dynamic yet. It becomes a whole lot more when you are able to ask the user to enter a value for variables. This is possible using the command *INPUT*. Take a look at the following example.

```
CLS
DIM FrontName AS STRING, BackName AS STRING
PRINT "Enter your front name: ";
INPUT FrontName
INPUT "Enter your back name: ", BackName
PRINT "So your name is "; FrontName; " "; BackName
END
```

You can clearly see the function of Input in this example, but not the syntax. I will discuss it soon. Anyway, input asks the user to input a value for the variable specified in the input command. As you can see, it is also able to display a prompt text, which is a string (variable) before the variable to ask the value of. If you don't specify a prompt text, a question will be used as prompt text (ugly), so I advice you to always enter a prompt text, and if you don't want one, not even a question mark, use an empty string "".

You can also ask values of variables of numerical type. However, if the user doesn't enter a number in that case, you get a 'Redo from Start'-error, which stands ugly in your *program, but allowing the user to type again. You'll later on (following tutorial?) learn on how to prevent this.*

`INPUT [prompt$,] var`
`INPUT [prompt$;] var`
	Asks the user to input a value for the variable var. You can specify a prompt text. If you do specify one, you can separate the var and prompt by either a comma or semicolon. A colon is the normal way, a semicolon causes a question mark to be printed after prompt.

CHAPTER PROGRAM

At the end of each programming chapter, there'll be a program written by myself, using most of the things you've learnt in this chapter, by means of convention. This program will be a bit bigger than the examples you've seen so far. Also, after each chapter program there are some exercises for you to solve. Try making these exercises before proceeding to the next chapter. The answers to all exercises will be at the end of the tutorial.

```
CLS
DIM NiceColor AS INTEGER

LOCATE 3, 3
PRINT “This program asks you to enter your favorite color.”
LOCATE 4, 3
PRINT “Enter the color number of your favorite color (0-15).”
LOCATE 5, 3
INPUT “Your favorite color: “, NiceColor
COLOR NiceColor
LOCATE 6, 3
PRINT “So your favorite color has color number”; NiceColor;
PRINT “!”
COLOR 7, 8
LOCATE 8, 3
PRINT “Press any key or wait 5 seconds to quit..”
SLEEP 5
CLS
END
```

Here are some exercises:

Exercise #1: Print the names of all 16 colors on the screen, in their respective color. E.g. ‘Green’ in green, ‘Red’ in red, etc.

Exercise #2: Ask the user to input his two most favorite colors, and then ask the user to input his/her front and back name. Clear the screen, then print the full name on one line, in which the front name is written in one favorite color, and the back name in the other.

Exercise #3: Make a program which poses the user 10 questions, with each question the screen will be cleared. At the end, print out, on one screen, what the user answered on all questions.

CHAPTER IV: STRUCTURE

In this chapter, structuring your code will be discussed, with topics like making choices and data modifiers. These are things you really need to know, because they are used *very* often. So study this chapter well.

IDENTING

In this chapter the main topic is structuring your code. A very important aspect of this structuring is identing. Identing is the addition of tabs to your code to make it more readable. The position of these tabs is usually when a separate code-section is created. All will become clear when you arrive at the section about choices. My advice to you is to get used to identing yourself, because it really makes your code better to understand. Now, first let’s go to the data modifiers.

DATA MODIFIERS

In QB there are functions (note *functions*) for accessing and retrieving data from strings, and for converting one datatype into an other.

Let’s first start with the *String Modifiers*.

STRING MODIFIERS

Run the following example to see some string modifiers in action:

```
INPUT “Enter your name: “, yourname$
PRINT “Your name in capitals: “; UCASE$(yourname$)
PRINT “Your name in lowercase: “; LCASE$(yourname$)
PRINT “Length of your name: “; LEN(yourname$)
PRINT “First character: “; LEFT$(yourname$, 1)
PRINT “Last character: “; RIGHT$(yourname$, 1)
PRINT “2nd character: “; MID$(yourname$, 2, 1)
INPUT “Enter spaces, a word, and again spaces: “, lstr$
PRINT “Clipped off first spaces: “; LTRIM$(lstr$)
PRINT “Clipped off last spaces: “; RTRIM$(lstr$)
PRINT “Here are 10 spaces: “; SPACE$(10)
INPUT “Enter a number: “, n%
PRINT “Stringed number:”; STR$(n%)
INPUT “Enter again a number: “, m$
PRINT “Numerical value:”; VAL(m$)
INPUT “Now enter 1 character: “, char$
PRINT “ASCII code of character:”; ASC(char$)
PRINT “20 of these chars: “; STRING$(20, char$)
INPUT “Now enter a value between 33 and 255: “, ascii%
PRINT “Corresponding ASCII character: “; CHR$(ascii%)
END
```

Phew, that was a whole lot. Let’s explain now.

`UCASE$(s$)`	Returns a string equivalent to the string s$ in upper case. (A-Z).
`LCASE$(s$)`	Returns a string equivalent to the string s$ in lower case. (a-z).
`LEN(s$)`	Returns the length of the string s$.
`LEFT$(s$, n%)`	Returns the first n% characters of the string s$.

`RIGHT$(s$, n%)`	Returns the last n% characters of the string s$.
`MID$(s$, p%, n%)`	Returns n% characters starting at character p% in string s$. The first character in the string s$ is number 1, the last is LEN(s$).
`LTRIM$(s$)`	Removes all spaces in front of s$ and returns this.
`RTRIM$(s$)`	Removes all spaces at the back of s$ and returns this.
`SPACE$(n%)`	Returns a string consisting of n% spaces.
`STR$(n%)`	Returns the string equivalent of number n%. E.g. a% = 5, then STR$(a%) will return “ 5”.
`VAL(s$)`	Returns the number equivalent of string s$. It’s like STR$ reversed.
`ASC(char$)`	Returns the ASCII Code of a character in char$.

`STRING$(n%, c$)`
`STRING$(n%, c%)`
	Generates and returns a string consisting of n% times the same character, c$. If instead of a string c$ you enter a number c% as the second parameter, this is the ASCII code of the character to duplicate.

`CHR$(ascii%)`	Returns the character connected with the ASCII code ascii%. It’s like ASC reversed.

I don’t assume anyone would understand this fully in the beginning, that’s why I advice you to practice a bit with the different Functions described above.

I’ll now go a little bit deeper into the subject of *Functions*.

First read the section Commands: Statements and Functions? on page 4 to begin with, if you hadn’t done it already.

If you would just type *LEFT$(“Hello”, 2)* in the QB IDE and try to run it, you’ll get an error. This is because Left is a function, and functions return values. It is required in QB to capture the values returned. Something has to be done with the values, a return value simply cannot be ignored. You could e.g. do *A$ = LEFT$(“Hello”, 2)* to store the return value of the function Left into a string A$ (which will obviously contain *“He”*). So, QB requires you to do something with the return value. On the other hand, statements do not have this since they do not return a value, e.g. the Print or Color statement. Try to keep this in mind.

Note that in C/C++, return values can be ignored.

Also, there is also a MID$ statement. See the following example.

```
A$ = “Hello World”
PRINT A$
MID$(A$, 8, 4) = “ario”
PRINT A$
```

I’m sure you know what I mean after running this example. It’s actually the MID$ function, but reversed. In this case, you don’t want to retrieve some characters from the string, but change some characters in the string. Further, it’s just the same as MID$ Function.

Now let’s change subject to other data modifiers.

CONVERSION FUNCTIONS

A very important part of data modifiers are the conversion functions. They are used to convert data from one datatype to another. Usually data gets lost during this operation. Some of these modifiers are also mathematical operators, and I will show them here as well. Since this is quite easy to understand for anyone who has little maths knowledge, I don’t give an example here. I’ll just go to the function table right away.

`CINT(number)`	Returns an integer that is number converted to integer. Note that when the decimal <= .5 then CINT will round towards zero, if decimal > .5 then CINT will round away from zero.
`CLNG(number)`	Same as CINT, only converts to long.
`CSNG(number)`	Converts a number to single precision and returns this.
`CDBL(number)`	Converts a number to double precision and returns this.
`INT(number)`	Converts to integer by always rounding down.
`FIX(number)`	Converts to integer by chopping off decimals.
`ABS(number)`	Gets the absolute value of number (erase sign).
`COS(number)`	Gets the cosine of number.
`SIN(number)`	Gets the sine of number.
`TAN(number)`	Gets the tangent of number.
`ATN(number)`	Gets the arctangent of number.
`SGN(number)`	Gets the sign of number. If number = 0 then SGN will return zero, if number < 0 then SGN will return –1, else if number > 0 then SGN will return 1.
`EXP(number)`	Gets e ^ number (e = 2.718281828…)
`LOG(number)`	Gets the natural logarithm of number (EXP inversed).
`SQR(pos.number)`	Gets the cube root of a positive number.
`HEX$(number)`	Gets the hexadecimal equivalent of number.
`OCT$(number)`	Gets the octal equivalent of number.
`CVI(s$)`	Binary conversion of a 2 byte-string to integer.
`CVL(s$)`	Binary conversion of a 4 byte-string to long.
`CVS(s$)`	Binary conversion of a 4 byte-string to single.
`CVD(s$)`	Binary conversion of a 8 byte-string to double.
`MKI$(number)`	Binary conversion of an integer to a 2 byte-string.
`MKL$(number)`	Binary conversion of a long to a 4 byte-string.
`MKS$(number)`	Binary conversion of a single to a 4 byte-string.
`MKD$(number)`	Binary conversion of a double to a 8 byte-string.

Everything except maybe the last 10 should be clear. Math-experts may already have noticed that QB doesn’t have arcsine or arccosine functions. That is because they can be derived from other routines. Anyone who doesn’t like maths, now go to the next page! ☺

Arcsine	Arcsin(x) = ATN(x / SQR(1 – x * x))
Arccosine	Arccos(x) = 2 * ATN(1) – ATN(x / SQR(1 – x * x))
Sine Hyperbolicus	Sinh(x) = (EXP(x) – EXP(-x)) / 2
Cosine Hyperbolicus	Cosh(x) = (EXP(x) + EXP(-x)) / 2
Tangent Hyperbolicus	Tanh(x) = (EXP(x) – EXP(-x)) / (EXP(x) + EXP(-x))
Areasine	Arcsinh(x) = LOG(x + SQR(x * x + 1))
Areacosine	Arccosh(x) = LOG(x + SQR(x * x – 1))
Areatangent	Arctanh(x) = LOG((1 + x) / (1 – x)) / 2
Secans	Sec(x) = 1 / COS(x)
Cosecans	Cosec(x) = 1 / SIN(x)
Contangent	Cotan(x) = 1 / TAN(x)
Arcsecans	Arcsec(x) = 2 * ATN(1) – SGN(x) * ATN(1 / SQR(x * x – 1))
Arccosecans	Arccosec(x) = SGN(x) * ATN(1 / SQR(x * x – 1))

Arccotangent	Arccotan(x) = 2 * ATN(1) – ATN(x)
Secans Hyperbolicus	Sech(x) = 2 / (EXP(x) + EXP(-x))
Cosecans Hyperbolicus	Cosech(x) = 2 / (EXP(x) – EXP(-x))
Cotangent Hyperbolicus	Cotanh(x) = (EXP(x) + EXP(-x)) / (EXP(x) – EXP(-x))
Areasecans	Arcsech(x) = LOG((SQR(1 – x * x) + 1) / x)
Areacosecans	Arccosech(x) = SGN(x) * LOG((SQR(x * x + 1) + 1) / SGN(x) * x)
Areatangent	Arccotanh(x) = LOG((x + 1) / (x – 1)) / 2
Logarithm	LogN(x, n) = LOG(x) / LOG(n)

Well, so far about the maths business, that’s why we have maths… The point I wanted to make here is that if you miss your own mathematical function, you can almost always make it yourself. Anyway, let’s get on with structuring!

There are still 8 conversion routines which no one ever uses (I think). These are CVIMBF, CVLMBF, CVSMBF, CVDMBF, MKIMBF$, MKLMBF$, MKSMBF$ and MKDMBF$. These are the same as their respective functions without ‘MBF’, but the only difference is that these work with (old?) Microsoft Binary Format. I’ve never used them… ☺

MAKING CHOICES PART ONE

In every programming language, making choices is an all-time thing you do. Here I’ll discuss the most popular way of making choices, namely IF … THEN. Take a look at the following example:

```
DIM Number AS INTEGER
INPUT “Enter a number: “, Number
IF Number = 0 THEN PRINT “Number equals zero”
IF Number <> 0 THEN PRINT “Number differs from zero”
```

This code should be clear when you pronounce the code for yourself. If number is zero, then print this on the screen. If number is not zero, then print that on the screen.

In QB, there is no character like ≠. Instead you should use <> (above comma and point on your keyboard) for not equal to.

In the previous example, the Ifs were inefficiently used. There are some features of IF I’ll discuss now. Take a look at this:

```
DIM Number AS INTEGER
INPUT “Enter a number: “, Number
IF Number = 0 THEN PRINT “Is zero” ELSE PRINT “Not zero”
```

This is the same example as previous, but now in three lines. An ELSE clause has been used in this example. Every IF can also contain *one* ELSE clause, which will be run if the condition in the IF statement is false. Obviously, the ELSE clause will not be run if the IF statement is true.

Another option of the IF … THEN is to have it execute multiple lines when true or false:

```
PRINT “Division program, calculates 1 / x”
INPUT “Enter x: “, x%
IF x% = 0 THEN
     PRINT “Division by zero is not allowed”
     PRINT “Quitting..”
ELSE
```

```
      PRINT “Division is allowed by x =”; x%
      PRINT “The answer is:”; CDBL(1# / x%)
END IF
```

This program asks the user to input a value to calculate 1 / x with. Since division by zero is not allowed, it checks if x = 0, and if so it prints a message and doesn’t calculate anything. If it is not zero, the calculation will be executed.

So, in this case, everything between IF and ELSE will be executed when the IF condition is true, and everything between ELSE and END IF will be executed when the IF condition is false. You can however also make IF statements on more lines without ELSE. ELSE is optional:

```
PRINT “Division program, calculates 1 / x”
INPUT “Enter x: “, x%
IF x% = 0 THEN
      PRINT “Division by zero is not allowed”
      PRINT “Quitting..”
      END
END IF
PRINT “Division is allowed by x =”; x%
PRINT “The answer is:”; CDBL(1# / x%)
```

Now the program will be automatically quit when x = 0, so it doesn’t even reach the division. The point is that ELSE is optional, as well as the thing we’ll discuss now: ELSEIF.

If you want to make an IF on multiple lines, an END IF clause is required to specify the end of the IF block, else QB won’t know what should or what shouldn’t be executed when IF is false or true. If you make an IF on one line, END IF is not to be used.

Now about ELSEIF, take a look at the example:

```
INPUT “Enter a number: “, n%
IF n% < 17 THEN
      PRINT “Number is below 17!”
ELSEIF n% = 17 THEN
      PRINT “Number equals 17!”
ELSE
      PRINT “Number is above 17!”
END IF
```

As you can see in this example, ELSEIF actually is a second IF. The only difference is that ELSEIF will only be checked for true or false if the IF is false. If the ELSEIF is also false, QB will then go to the ELSE, only then. The advantage of ELSEIF is that you can use large numbers of them in one if.

To see the difference between using 2 IF’s and using one IF and one ELSEIF, see the following example.

```
INPUT “Enter a number: “, n%
PRINT CHR$(13) + CHR$(10)
PRINT “This is the output of the 2 IFs way:”
IF n% < 10 THEN PRINT “Smaller than 10”
IF n% < 25 THEN PRINT “Smaller than 25”
PRINT CHR$(13) + CHR$(10)
PRINT “This is the output of the IF + ELSEIF way:”
IF n% < 10 THEN
```

```
     PRINT “Smaller than 10”
ELSEIF n% < 25 THEN
     PRINT “Smaller than 25”
END IF
```

When you run this and enter a number like 7, you’ll see the difference clearly ☺.

If you want an IF statement on one line, you cannot use ELSEIF. ELSEIF can only be used in multiple-line IF statements.

`IF … THEN`	This is the general IF clause. If the condition on the points is true, then the code after THEN is executed. If it is false, if goes on to any elseif or else if there.
`ELSEIF … THEN`	To extend an IF clause with more options, this is just like IF.
`ELSE`	To extend an IF clause with a general option, this clause will only be executed if no other clause is true.
`END IF`	Denotes the end of an IF block.

Making Choices Part Two

If you have 100 options for an IF clause, you’ll have to type like hell to implement all 100 options. An other, and overall better choice making command is SELECT. It is less typing, keeps better structure in your program, is easy to use and is sometimes faster. Here’s an example:

```
INPUT “Enter a number: “, n%
SELECT CASE n%
     CASE 1, 2, 3
          PRINT “It is one, two or three”
     CASE 4 TO 6
          PRINT “It is in the range 4-6”
     CASE IS > 6
          PRINT “Larger than three”
     CASE IS < 0
          PRINT “Smaller than zero”
     CASE ELSE
          PRINT “It is zero”
END SELECT
```

As you can see in this example, with select it is easy to add multiple options at once. The structure of the select block is as follows. To open the block you type *SELECT CASE* and thereafter the variable to check. Then you add *CASE*s with thereafter the options the variable may have. You can do option comma option etc, or you could do option TO option. With IS you can make greater/smaller equations, and finally there’s ELSE if no other case was true. It’s easy to understand after a couple of practices.

Only one CASE can be executed per SELECT block, just like the IF-ELSEIF construction.

Just like with IF, select is automatically multiple lines, which means everything between the CASE keywords (or END SELECT) will be executed when the specific condition is true. Now a small summary.

`SELECT CASE var`	Starts the select block with variable var to check.

`CASE option [, option, ……]`
`CASE IS expression`
`CASE range1 TO range2`
`CASE ELSE`
	These are the possible case clauses. Firstly you can add as many options as you want. The case will be true when one or more of these options *exactly* equal the variable (keep this in mind when using floating point variables!). Secondly, you can add an expression to a case by means of IS. If the expression is true, the case will be executed. Thirdly, you can specify a range in which the variable can be. If the variable is in the range, the case is true. Fourthly, when all other cases are false, the case else will be executed.

`END SELECT`	This denotes the end of a select block.

JUMPING AROUND

The last paragraph in this chapter is about jumping. Not much experienced programmers use jumping though, only when it's really necessary. That's also the point, you still need to know it. Also, I'll explain it now because many beginning programmers use it a lot. ☺ Here's an example on how to use it:

```
Again:
PRINT “Hello”; CHR$(13); “Please enter you name: “;
INPUT “”, yourname$
IF yourname$ = “” THEN
     PRINT “You entered nothing!”; CHR$(13)
     GOTO Again
END IF
PRINT “So your name is “; yourname$
TypeAgain:
PRINT “Is this correct (Y/N)?”
INPUT “”, YESNO$
SELECT CASE UCASE$(YESNO$)
     CASE “N”
          GOTO Again
     CASE “Y”
          GOTO EndNow
     CASE ELSE
          GOTO TypeAgain
END SELECT

EndNow:
END
```

As you can see in this program, GOTO can be used to control the flow of your program. With GOTO, you can specify a *label* and the program will go to the label in your program. A label is a name with a colon behind it on a separate line. Your program will then just continue normally from the label.

`Label:`	A label in your program. Must consist of normal characters.
`GOTO Label`	This statement makes the program jump to Label. From there, the program will run further.

If you specify a label in Goto that doesn’t exist, you’ll get an error.

The main disadvantage of using goto’s is that the more goto’s you use, the more your code will look like spaghetti. That’s why unreadable, criss-cross linked programs are called programs with spaghetti code. Therefore they invented another jumping statement, which is able to induce less spaghetti ☺. This statement is *GOSUB*. Here’s an example:

```
PRINT “ABC Formula.”
A! = 0
B! = 0
C! = 0
N% = 0
X1! = 0
X2! = 0
GOSUB GetValues
GOSUB Calculate
GOSUB PrintForm
END

GetValues:
INPUT “Enter A: “, A!
INPUT “Enter B: “, B!
INPUT “Enter C: “, C!
RETURN

PrintForm:
PRINT “The function:”
PRINT A!; “ * x ^ 2 ”;
IF SGN(B!) = 1 THEN PRINT “+”; ELSE PRINT “-“;
PRINT ABS(B!); “ * x “;
IF SGN(C!) = 1 THEN PRINT “+“; ELSE PRINT “-“;
PRINT ABS(C!)
PRINT “Has”; N%; “ intersections with the x-axis”
IF N% >= 1 THEN PRINT “X1 =”; X1!
IF N% = 2 THEN PRINT “X2 =”; X2!
RETURN

Calculate:
D# = B! ^ 2 – 4 * A! * C!
IF D# < 0 THEN
     N% = 0
ELSEIF D# = 0 THEN
     N% = 1
     X1! = (-B! + SQR(D#)) / (2 * A!)
ELSEIF D# > 0 THEN
     N% = 2
     X1! = (-B! – SQR(D#)) / (2 * A!)
     X2! = (-B! + SQR(D#)) / (2 * A!)
END IF
RETURN
```

This quite large example is actually the ABC Formula, taught on school in Maths classes of… what is it? Year 3? Anyway, it's maths ☺. In the above example, Gosub is used to do various sections of the program in a separate part of the program. That's why Gosub is some better at this than Goto; you can more easily read it. Unlike Goto, Gosub requires a Return at the end of the block to jump to. This is the great difference with Goto. Goto just jumps to a label and runs the program further from there. Gosub jumps to a label, runs the program normally, until it encounters a Return statement. The program will then jump back to the place of the last Gosub, in other words, the Gosub that jumped to the certain section. Then the program will run further. Try to keep this in mind and look again at the example. Now you should see why it still runs as it should! ☺.

`GOSUB Label`	Jumps to Label and executes code from there until it reaches a Return statement, then the program will contain after the Gosub statement.
`RETURN`	This statement causes the program to return to the last Gosub called.

Avoid the nesting of Gosub statements. This will eventually result in an 'Out of Stack Space' error, which will immediately halt your program.

Nesting is consequently putting blocks of code in other blocks of code, etc. E.g. putting an IF in an IF, which also is in an IF, etc. Nesting can be done with various commands like Gosub, If, Select and later on you'll learn it can also be done with a.o. loops and procedures.

I advice you to put your Labels referred to by Gosub at the end of your program, after an end. This way they'll never be run without Gosub calling it.

CHAPTER PROGRAM

Ok, this chapter program is about palindromes, which are words with a special feature. Viz. that when they are reversed, it still is the same. E.g. RADAR is a palindrome. Here the program comes:

```
DIM Original AS STRING, Reversed AS STRING

Again:
CLS
PRINT “@@@@@@@@@@@@@@@@@@”
PRINT “ @@@@@@@@@@@@@@@@”
PRINT “  Palindromotron”
PRINT “ @@@@@@@@@@@@@@@@”
PRINT “@@@@@@@@@@@@@@@@@@”

AgainTypeWord:
PRINT CHR$(13); “Enter a word below to see if it is”
INPUT “a palindrome: “, Original
IF LEN(Original) = 0 THEN GOTO AgainTypeWord
Original = UCASE$(LTRIM$(RTRIM$(Original)))

GOSUB ReverseWord

PRINT “The word “; Original;
IF Original = Reversed THEN
     PRINT “ is a palindrome”
```

```
ELSE
     PRINT “ is not a palindrome”
END IF

AgainTypeYesOrNo:
PRINT CHR$(13); “Do you want to enter another word?”
INPUT “Y/N: “, YESNO$
YESNO$ = UCASE$(LTRIM$(RTRIM$(YESNO$)))
SELECT CASE YESNO$
     CASE “Y”
          GOTO Again
     CASE “N”
          CLS
     CASE ELSE
          GOTO AgainTypeYesOrNo
END SELECT

END



ReverseWord:
‘this is a crappy inefficient algorithm, but since I have
‘to use goto.. lol ;)
Reversed = “”
CurrentPosition% = LEN(Original)
NextPosition:
Reversed = Reversed + MID$(Original, CurrentPosition%, 1)
CurrentPosition% = CurrentPosition% - 1
IF CurrentPosition% > 0 THEN GOTO NextPosition
RETURN
```

Now a few exercises to remember what you saw in this chapter… ☺

Exercise #1: Make a kind of quiz with about 20 questions. At the end of the questions, show how many the player guessed wrong.

Exercise #2: Ask the user to input a whole english sentence. Then ask the user to input a letter (a till z). The program outputs the number of those letters found it the sentence, either lower case or capital.

Exercise #3: Ask the user for as much numerical values as he likes to. Then, calculate the average and sum of all these values and show them to the user.

Exercise #4 (harder): Show a character on the screen. Then ask the user what way it should go. (Ask for values like ‘left’, ‘right’, ‘up’ and ‘down’). Then move the character on the screen in the direction the user entered. Let the user be able to walk around the screen if he wants to, i.e. the program has to repeat itself after moving the character. Exit the program when the user types something like ‘exit’ or ‘quit’. Also make sure the character cannot move out of the screen (you’ll get an error then ☺).

CHAPTER V: LOOPS AND BOOLEANS

BOOLEAN OPERATORS

Boolean operators are operators that can paste two conditions together. You have used boolean operators often in normal life, even without knowing it. Take a look at the following example about the usage of boolean operators. In the example, you try to look for a person by checking his ASL.

```
IF PersonName$ = “Greg” THEN
     IF PersonAge% = 27 THEN
          IF PersonHabitat$ = “New York” THEN
               ‘ you found him!
          END IF
     END IF
END IF
```

As you can see, this code doesn’t look very clean, even with *identing*. That’s why they invented boolean operators. Boolean operators are operators like AND, OR and NOT. See what the example will be with boolean operators:

```
IF (PersonName$ = “Greg”) AND (PersonAge% = 27) AND
(PersonHabitat$ = “New York”) THEN
     ‘found him!
END IF
```

In this code, you can immediately see what all conditions are. Also, you may know get to realize you used boolean operators often before… ☺. There are more boolean operators than AND. I will now show you a TF Table, what stands for True-False Table. In this table, you can see what the result of two conditions pasted together with the specific boolean operator will be. It might be hard to understand, therefore I'll not go talking to empty space, and here they come:

AND	**True**	**False**
True	True	False
False	False	False

OR	**True**	**False**
True	True	True
False	True	False

XOR	**True**	**False**
True	False	True
False	True	False

EQV	**True**	**False**
True	True	False
False	False	True

IMP	**True**	**False**
True	True	False
False	True	True

	True	**False**
NOT	False	True

In these TF Tables, the vertical row of true and false is the first (left) condition. The horizontal row is the second (right) condition. Well, what does all this stuff mean? Lets take the common AND operator for example. If you see in the table, AND only returns true when the first condition (vertical) is true and when the second condition (horizontal) is also true. Of course, you already knew this ☺ Here’s some sample code to check when the boolean operators will be run:

```
A = 1
B = 2
C = 3
```

```
D = 4

IF (A = 1) AND (B = 2) THEN
     ‘this IF condition will be true when A = 1 and B = 2.
     ‘so when A = 1 and B = 2
     ‘if any condition is false, the IF will be false.
     ‘(AND is the logical conjunction operator)
END IF

IF (B = 2) OR (C = 3) THEN
     ‘this IF condition will be true when B = 2 or C = 3.
     ‘that means: it will be true when either B = 2 or C = 3
     ‘or when both conditions are true:
     ‘TRUE WHEN:
     ‘B = 2 and C <> 3
     ‘B <> 2 and C = 3
     ‘B = 2 and C = 3
     ‘so OR is very flexible ☺
     ‘(OR is the Inclusive OR operator)
END IF

IF (C = 3) XOR (D = 4) THEN
     ‘this IF condition will be true when one of both
     ‘conditions is false, and one is true. So:
     ‘TRUE WHEN:
     ‘C = 3 and D <> 4
     ‘C <> 3 and D = 4
     ‘(XOR is the Exclusive OR operator).
END IF

IF (D = 4) EQV (A = 1) THEN
     ‘this IF condition will be true when both conditions
     ‘are true or when both are false. So:
     ‘TRUE WHEN:
     ‘D = 4 and A = 1
     ‘D <> 4 and A <> 1
     ‘(EQV is the logical equivalence operator)
END IF

IF (A = 1) IMP (B = 2) THEN
     ‘this IF condition will be true when the second condition
     ‘is false and the first one is true. So:
     ‘TRUE WHEN:
     ‘A = 1 and B = 2
     ‘A <> 1 and B = 2
     ‘A <> 1 and B <> 2
     ‘I never use IMP though
     ‘(IMP is the logical implication operator)
END IF
```

If you’ve studied all six TF Tables, you should have noticed that the NOT operator doesn’t accept two conditions. This is due to the fact that And, Or, Xor, Eqv and Imp are binary operators and Not is a unary operator. This means that Not accepts one parameter. Not turns around the value of this parameter. In other words, True becomes False and False becomes True. Take a look:

```
IF NOT(A = 5) THEN
     ‘this code will be run when A is not five
END IF
```

Looks simple, but keep in mind Not can also contain series of multiple conditions connected to eachother with other binary operators.

If you use multiple binary operators in one whole condition, you should place parenthesis to show to QB which condition to evaluate first. If you don’t know what I mean with ‘evaluate first’, think of just mathematical arithmetic:

A = 7 * 8 / 3 + 2 * 4 / 9 – 3 ^ 2 * 5

Most logical errors in programs occur this way. You should put parenthesis about sections of code that have to be evaluated first. The above mathematical assignation can be e.g. like this:

A = (7 * 8 / (3 + 2) * 4 / (9 – 3)) ^ (2 * 5)

This will get a whole lot different answer than the previous one. So if your condition gets a bit complex, put parenthesis:

```
IF ((Score > 500 AND Skill < 4) OR (Score > 100 AND Skill >
9)) EQV (NOT(EnemiesKilled < 100) IMP (EnemiesKilled > 10 XOR
BossesKilled < 2)) THEN
     ‘you may find out when this will be run! ☺
END IF
```

LOOP DA LOOP WITH FOR AND NEXT

Now we finally are at the loops. I’ll start with the loop most difficult to learn, but very often used. This is the For loop. The For loop is a counter loop used in almost anything. You specify a lower and higher range, a step factor and For will count with your steps from the lower range till the higher range has been reached. Here’s an example:

```
FOR counter% = 1 TO 100 STEP 1
     PRINT counter%
NEXT counter%
```

As you can see when you run this example, the numbers 1 up to and including 100 are printed on the screen. I’ll now explain how this is possible. The first line defines a variable counter%, which gets the value 1 (the value before TO). QB will then run your program further, until it reaches Next. Then QB will jump back to the For statement, and increase counter% by your step factor, in this case 1. It will then loop again, increase again, loop again, etc. This process continues until counter% has exceeded the value of the high range (after TO), so when counter% > 100. QB will then continue with your program after Next.

I understand if this is hard for you to understand, but this loop is mostly used and by practising with it a bit you will get to know much more about it.

`FOR var = begin TO end [STEP m]`	This statement announces the beginning of a counting loop. At start, var will be equal to begin. After each loop, var will be increased with m. The loop ends when var has exceeded

	the range begin-end. Note that the stepfactor is optional. When left away, the stepfactor will always be 1 (one).
`NEXT [var]`	This is the end of a counting loop. When reached here, the program will go back to the FOR var to continue the loop, or just skip this statement when the loop has finished.

For is really used for doing very many things, here is another example of For:

```
INPUT “Enter your name: “, yourname$
FOR I% = 1 TO LEN(yourname$)
     PRINT MID$(yourname$, I%, 1)
NEXT I%
```

Note that when begin immediately exceeds end at the start of a loop, the For loop is never run. So For M = 1 TO 0 STEP 1 will never be looped.

LOOP DA LOOP WITH DO AND LOOP

The second loop commands are also very often used, and should therefore be part of your knowledge as well. The loop commands are Do and Loop. A Do loop can be two things; either an infinite loop, or a loop that waits for a certain condition. Here’s the infinite loop:

```
DO
     PRINT “Infinite loop!”
LOOP
```

You can end a program running in QB at any time by pressing [Ctrl] + [Break] simultaneously. If you have compiled your program into an EXE, this will not be possible unless you have selected the option ‘Produce Debug Code’.

As you can see in the example, the text ‘Infinite Loop!’ is printed over and over again, until you force the program to stop with Ctrl+Break. The loop theory is the same as For, but I’ll still explain it here. When the program reaches Do, it means the beginning of a new loop. It then executes code until it reaches Loop. The program will then jump back to Do and runs the cycle again. This will never stop (infinite loop) unless your program says so (see *Exit* further in the chapter).

Then there was the second possibility a Do loop could be, viz. the loop that waits for a condition. Here’s an example:

```
DIM I AS INTEGER
I = 0
DO
     I = I + 1
     PRINT I
LOOP UNTIL (I = 100)
PRINT “First loop ended”

I = 0
DO WHILE I < 100
     I = I + 1
     PRINT I
```

```
LOOP
PRINT “Loops ended”
```

As you can see, the keywords UNTIL and WHILE can be used to accept a condition. Until causes the loop to loop *until* the expression is true. While causes the loop to loop *while* the expression is true, and it will stop when the expressions turns false. Of course, you can combine multiple conditions using any or more of the boolean operators. But you already knew that didn’t you? ☺

There is however a difference between putting the Until or While at the Do or at the Loop. Here’s an example that illustrates this difference:

```
I% = 1
DO
     PRINT “We are in loop one”
LOOP UNTIL I% = 1

DO UNTIL I% = 1
     PRINT “We are in loop two”
LOOP
```

As you can see, your program only prints we are in loop one on the screen. This is because when placing the condition at the LOOP, the loop will be run at least one time, because checking for the condition takes place after each loop. If you place the condition at the DO, the condition will be checked before each loop takes place. So if the condition is until true anyway, the loop won’t be run and the program will skip it right away, continuing at the LOOP.

To get used to Do Loop, try experimenting with it! ☺

`DO [WHILE|UNTIL condition]`	Marks the beginning of a Do loop. When a condition is placed at the Do, this condition will be checked before running each loop. Therefore it is possible the loop will never be run.
`LOOP [WHILE|UNTIL condition]`	Marks the end of a Do loop. From here the programs jumps back to the corresponding Do. That is, if the condition attached to Loop, if any, is true. If a condition is placed at Loop, this condition is checked after running each loop.

You cannot use two conditions in a Do Loop. I.e. you can only place one Until or one While in the loop, either at Do or at Loop. Both having a condition is not possible.

LOOP DA LOOP WITH WHILE AND WEND

This is the last of the loops, not often used since it is actually from an older version of BASIC. However, I’ll still explain it to you, because some people still use them. I advice you to use Do Loop instead though. Here’s an example on how to use While loops:

```
I% = 0
WHILE I% < 100
     I% = I% + 1
     PRINT I%
WEND
```

I don’t have to explain it actually, as it is the same as a Do While Loop (i.e. DO WHILE condition ……… more stuff ……… LOOP). I think the only advantage of a While loop above a Do While Loop is that it is slightly less typing… ☺

Well, that were all loops for now… ☺

`WHILE condition`	Marks the beginning of a while loop. The condition attached to While will be checked before each loop takes place. Therefore, it is possible the loop will never be run.
`WEND`	Marks the end of a while loop. From here, the program jumps back to the corresponding While.

Exit the Loop da Loop

As already implied previously, there is also a command to manually stop loops, even if they are infinite or waiting for a condition to be true. This command is *EXIT* and is often used. After exit you can specify what kind of loop to stop (*DO* or *FOR*) and your program will then exit *the first found* loop of that type. Note that it only stops One loop, and not all loops of that type. Here’s an example of Exit:

```
I% = 0
DO
     I% = I% + 1
     IF I% = 500 THEN EXIT DO
LOOP
PRINT “Stopped the DO loop at I% =”; I%

FOR I% = 1 TO 500
     IF I% = 250 THEN EXIT FOR
NEXT I%
PRINT “Stopped the FOR loop at I% =”; I%
```

You should know that if in this example EXIT hadn’t been executed, the first loop would have looped forever, until you would get an overflow of the integer (at 32767, viz. the maximum value of an integer). The second loop (if the program ever got there ☺) would have run till I% was 501. Try it…!

`EXIT DO`
`EXIT FOR`
	Immediately stops a loop. It stops the first found loop of the type you specified, if any.

Chapter Program

This chapter program is about prime numbers. Prime numbers are numbers that have no dividers. E.g. 13 is a prime number, as it can’t be divided by any number except 1 and itself. Here it comes:

```
DO
     CLS
     PRINT “Prime number calculator”
     PRINT “How many prime numbers should I calculate?”
     DO
          INPUT “”, N%
     LOOP WHILE N% < 1

     PRINT
```

```
    PRINT “ 2”
    PrimesFound% = 1
    Check% = 3
    DO UNTIL PrimesFound% = N%
        Dividable% = 0
        FOR prim% = 2 TO INT(SQR(Check%) + 1)
            IF INT(Check% / prim%) = Check% / prim% THEN
                Dividable% = -1
            END IF
        NEXT prim%
        IF Dividable% = 0 THEN
            PRINT Check%
            PrimesFound% = PrimesFound% + 1
        END IF
        Check% = Check% + 2 ‘only odd numbers
    LOOP

    PRINT “This were”; N%; “prime numbers”
    PRINT “Do you want to calculate some more?”
    DO
        INPUT “(yes or no): “, YESNO$
        YESNO$ = UCASE$(LTRIM$(RTRIM$(YESNO$)))
    LOOP UNTIL (YESNO$ = “YES”) OR (YESNO$ = “NO”)
    IF YESNO$ = “NO” THEN EXIT DO
LOOP
END
```

Now again for some cool exercises… ☺

Exercise #1: Make a program that can calculate the factorial of a positive number the user inputs. The factorial of a number is itself multiplied with all of its predecessors. E.g. the factorial of 6 = 1 * 2 * 3 * 4 * 5 * 6 = 720.

Exercise #2: (easy). Make a program that can calculate the sum of all numbers preceding the number, from 0. So with an input of e.g. 7 the output will be 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.

Exercise #3: (easy). Make a program that asks the user for a number. Then display a triangle on the screen which is number characters wide at the base. Each line 2 characters on the sides disappear.

Exercise #4: (harder). Make a program that is able to calculate all combinations of numbers within a specific range and length defined by the user. So if the user tells your program to show all combinations with length 4 and range 2, the program should output 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, and that this are 16 combinations. Have the maximum range be 10 and the maximum length be 8.

APPENDIX A

CATALOGUE

A list of all commands and keywords used in this part of QB Now!

Keyword/Command	**Page**
`ABS`	12
`AND`	20
`AS`	8
`ASC`	11
`ATN`	12
`CASE`	16
`CHR$`	11
`CDBL`	12
`CINT`	12
`CLNG`	12
`CLS`	6
`COLOR`	5
`COS`	12
`CSNG`	12
`CVD`	12
`CVI`	12
`CVL`	12
`CVS`	12
`CVDMBF`	13
`CVIMBF`	13
`CVLMBF`	13
`CVSMBF`	13
`DIM`	8
`DO`	24
`DOUBLE`	3
`ELSE`	15
`ELSEIF`	15
`END`	5
`END IF`	15
`END SELECT`	16
`EQV`	20
`EXIT`	25
`EXP`	12
`FIX`	12
`FOR`	22
`GOSUB`	18
`GOTO`	16
`HEX$`	12
`IF`	15
`IMP`	20
`INPUT`	9

INT	12
INTEGER	3
IS	16
LCASE$	10
LEFT$	10
LEN	10
LOCATE	6
LOG	12
LONG	3
LOOP	24
LTRIM$	11
MID$	11
MKD$	12
MKI$	12
MKL$	12
MKS$	12
MKDMBF$	13
MKIMBF$	13
MKLMBF$	13
MKSMBF$	13
NEXT	23
NOT	20
OCT$	12
OR	20
PRINT	5
REM	4
RETURN	18
RIGHT$	11
RTRIM$	11
SELECT	16
SGN	12
SIN	12
SINGLE	3
SLEEP	6
SPACE$	11
SQR	12
STEP	22
STR$	11
STRING	3
STRING$	11
TAN	12
THEN	15
TO (FOR)	22
TO (SELECT)	16
UCASE$	10
UNTIL	24
VAL	11
WEND	25

AFTERWORD

QB Now!
Part I: The Basics
By Neo
a.k.a. Neo Deus Ex Machina
HAR-SoftWare
http://www.harsoft.tk
hendrikantoniorecinos@hotmail.com
2nd of September 2004

Some cool QB sites with forums I visit:
http://www.qbnz.com
http://www.petesqbsite.com
http://www.qbasicnews.com

Thanks to these special people:
Oracle
Zack
Pete
Adigun Azikiwe Polack
Moneo
Na_Th_An
TurboFX
Akooma
Elcalen
Wildcard
Dav
Plasma
Relsoft
Piptol
Jocke The Beast
Josiah
Toonski (a.k.a. Jofers)
Zap
Sumo Jo
TheBigBasicQ
HQSneaker
And anyone else at the QBNZ & QBN forumboards

THANKS FOR READING THIS TUTORIAL

APPENDIX B

My answers to all exercises in this tutorial.

CHAPTER III, EXERCISE #1

Exercise #1: Print the names of all 16 colors on the screen, in their respective color. E.g. ‘Green’ in green, ‘Red’ in red, etc.

```
COLOR 0
PRINT “Black”
COLOR 1
PRINT “Blue”
COLOR 2
PRINT “Green”
COLOR 3
PRINT “Cyan”
COLOR 4
PRINT “Red”
COLOR 5
PRINT “Magenta”
COLOR 6
PRINT “Brown”
COLOR 7
PRINT “Light Grey”
COLOR 8
PRINT “Dark Grey”
COLOR 9
PRINT “Light Blue”
COLOR 10
PRINT “Light Green”
COLOR 11
PRINT “Light Cyan”
COLOR 12
PRINT “Light Red”
COLOR 13
PRINT “Light Magenta”
COLOR 14
PRINT “Yellow”
COLOR 15
PRINT “White”
```

CHAPTER III, EXERCISE #2

Exercise #2: Ask the user to input his two most favorite colors, and then ask the user to input his/her front and back name. Clear the screen, then print the full name on one line, in which the front name is written in one favorite color, and the back name in the other.

```
DIM FirstFav AS INTEGER, SecondFav AS INTEGER
INPUT “What’s your favorite colornumber? (0-15) “, FirstFav
INPUT “What your 2nd favorite colornumber? (0-15) “, SecondFav
INPUT “What’s your front name? “, FrontName$
INPUT “What’s your back name? “, BackName$
CLS
COLOR FirstFav
PRINT FrontName$; “ “;
COLOR SecondFav
PRINT BackName$
```

CHAPTER III, EXERCISE #3

Exercise #3: Make a program which poses the user 10 questions, with each question the screen will be cleared. At the end, print out, on one screen, what the user answered on all questions.

```
DIM Q1$, Q2$, Q3$, Q4$, Q5$, Q6$, Q7$, Q8$, Q9$, Q0$
CLS
INPUT “What does HTML stand for? “, Q0$
CLS
INPUT “What does QBASIC stand for? “, Q1$
CLS
INPUT “What does PHP stand for? “, Q2$
CLS
INPUT “What does CLS do? “, Q3$
CLS
INPUT “What does INPUT do? “, Q4$
CLS
INPUT “What was the predecessor of C? “, Q5$
CLS
INPUT “What is the PRINT equivalent in C? “, Q6$
CLS
INPUT “Did you understand this chapter (3)? “, Q7$
CLS
INPUT “Did you answer ‘NO’ on the previous question? “, Q8$
CLS
INPUT “Now, do you want to see your answers? “, Q9$
PRINT “Muhahaha, I’ll show them anyway ;)”
SLEEP 1
PRINT “What does HTML stand for?”
PRINT Q0$
PRINT “What does QBASIC stand for? “
PRINT Q1$
PRINT “What does PHP stand for? “
PRINT Q2$
PRINT “What does CLS do? “
PRINT Q3$
PRINT “What does INPUT do? “
PRINT Q4$
PRINT “What was the predecessor of C? “
PRINT Q5$
PRINT “What is the PRINT equivalent in C? “
PRINT Q6$
PRINT “Did you understand this chapter (3)? “
PRINT Q7$
PRINT “Did you answer ‘NO’ on the previous question? “
PRINT Q8$
PRINT “Now, do you want to see your answers? “
PRINT Q9$
SLEEP
END
```

CHAPTER IV, EXERCISE #1

Exercise #1: Make a kind of quiz with about 20 questions. At the end of the questions, show how many the player guessed wrong. (I’ll do 5 questions here… lol).

```
DIM Correct AS INTEGER, Wrong AS INTEGER
DIM Question AS STRING, Answer AS STRING
Correct = 0
Wrong = 0

Question = “What does HTML stand for?”
Answer = “hypertext markup language”
GOSUB PoseQuestion

Question = “What does PHP stand for?”
Answer = “php hypertext protocol”
GOSUB PoseQuestion

Question = “Are there already harddisks of 1024 GigaByte?”
Answer = “yes”
GOSUB PoseQuestion

Question = “Do you like QB?”
Answer = “yes”
GOSUB PoseQuestion

Question = “Did you crib to make this exercise?”
Answer = “no”
GOSUB PoseQuestion

CLS
PRINT “You answered”; Wrong; “questions incorrectly”
PRINT “You answered”; Correct; “questions correctly”
SLEEP
END

PoseQuestion:
CLS
AgainQuestion:
INPUT Question + “ “, Q$
Q$ = LCASE$(LTRIM$(RTRIM$(Q$)))
IF LEN(Q$) = 0 THEN GOTO AgainQuestion
IF Q$ = Answer THEN
     Correct = Correct + 1
ELSE
```

```
     Wrong = Wrong + 1
END IF
RETURN
```

CHAPTER IV, EXERCISE #2

Exercise #2: Ask the user to input a whole english sentence. Then ask the user to input a letter (a till z). The program outputs the number of those letters found it the sentence, either lower case or capital.

```
Nothing:
INPUT “Enter a sentence: “, sentence$
GoodSentence$ = LCASE$(LTRIM$(RTRIM$(sentence$)))
IF LEN(GoodSentence$) = 0 THEN GOTO Nothing
AgainNothing:
INPUT “Enter one letter (a-z): “, letter$
Letter$ = LCASE$(letter$)
IF LEN(letter$) <> 1 THEN GOTO AgainNothing
IF letter$ < “a” OR letter$ > “z” THEN GOTO AgainNothing

N% = 0
GOSUB CountLetters
PRINT “The sentence:”
PRINT sentence$
PRINT “Contains”; N%; letter$; “’s”
SLEEP
END

CountLetters:
NowPos% = 1
NextPos:
IF MID$(GoodSentence$, NowPos%, 1) = letter$ THEN N% = N% + 1
NowPos% = NowPos% + 1
IF NowPos% <= LEN(GoodSentence$) THEN GOTO NextPos
RETURN
```

CHAPTER IV, EXERCISE #3

Exercise #3: Ask the user for as much numerical values as he likes to. Then, calculate the average and sum of all these values and show them to the user.

```
DIM Sum AS SINGLE, Value AS SINGLE, Values AS INTEGER
Values = 0
Sum = 0

AnotherValue:
INPUT “Please enter a number: “, Value
Values = Values + 1
Sum = Sum + Value
AgainQ:
INPUT “Do you want to enter another number? (Y/N) “, YESNO$
IF UCASE$(YESNO$) = “Y” THEN
     GOTO AnotherValue
```

```
ELSEIF UCASE$(YESNO$) = “N” THEN
     ‘nothing
ELSE
     GOTO AgainQ
END IF

PRINT “You entered”; Values; “values”
PRINT “The sum is”; Sum
PRINT “The average is”; Sum / CSNG(Values)
SLEEP
END
```

CHAPTER IV, EXERCISE #4

Exercise #4 (harder): Show a character on the screen. Then ask the user what way it should go. (Ask for values like ‘left’, ‘right’, ‘up’ and ‘down’). Then move the character on the screen in the direction the user entered. Let the user be able to walk around the screen if he wants to, i.e. the program has to repeat itself after moving the character. Exit the program when the user types something like ‘exit’ or ‘quit’. Also make sure the character cannot move out of the screen (you’ll get an error then ☺).

```
X% = 40
Y% = 12

Again:
GOSUB MakeScreen
LOCATE 2, 1
INPUT “-> “, ACTION$
ACTION$ = UCASE$(LTRIM$(RTRIM$(ACTION$)))
SELECT CASE ACTION$
     CASE “LEFT”
          IF X% > 1 THEN X% = X% - 1
     CASE “RIGHT”
          IF X% < 80 THEN X% = X% + 1
     CASE “UP”
          IF Y% > 4 THEN Y% = Y% - 1
     CASE “DOWN”
          IF Y% < 25 THEN Y% = Y% + 1
     CASE “QUIT”, “EXIT”, “HALT”, “STOP”
          GOTO EndNow
     CASE ELSE
          LOCATE 2, 1
          PRINT “Invalid input”
          SLEEP 3
END SELECT
GOTO Again

EndNow:
END

MakeScreen:
CLS
```

```
LOCATE 3, 1
PRINT STRING$(80, 205)
LOCATE Y%, X%
IF Y% >= 24 THEN PRINT CHR$(1); ELSE PRINT CHR$(1)
LOCATE 1, 1
PRINT “Enter left, right, up, down or quit”
RETURN
```

CHAPTER V, EXERCISE #1

Exercise #1: Make a program that can calculate the factorial of a positive number the user inputs. The factorial of a number is itself multiplied with all of its predecessors. E.g. the factorial of 6 = 1 * 2 * 3 * 4 * 5 * 6 = 720.

```
DO
     INPUT “Enter a number (0 <= n <= 170): “, n%
LOOP UNTIL n% >= 0 AND n% <= 170
PRINT n%; “! =”;
FAC# = 1
FOR I% = 1 TO n%
     FAC# = FAC# * CDBL(I%)
NEXT I%
PRINT FAC#
```

CHAPTER V, EXERCISE #2

Exercise #2: (easy). Make a program that can calculate the sum of all numbers preceding the number, from 0. So with an input of e.g. 7 the output will be 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.

```
DO
     INPUT “Enter a number (0 <= n <= 1 bln): “, n&
LOOP UNTIL n& >= 0 AND n& <= 1000000000&

Sum# = 0
FOR i& = 1 TO n&
     Sum# = Sum# + CDBL(i&)
NEXT i&
PRINT “Total sum is”; Sum#
‘note that this is equal to Sum# = 0.5 * n& * (n& + 1)
```

CHAPTER V, EXERCISE #3

Exercise #3: (easy). Make a program that asks the user for a number. Then display a triangle on the screen which is number characters wide at the base. Each line 2 characters on the sides disappear.

```
DO
     INPUT “Enter base width (0 < n < 40): “, n%
LOOP UNTIL n% > 0 AND n% < 40
CLS
Length% = n%
FOR I% = 1 TO INT(n% / 2 + 1)
     LOCATE I%, I%
     PRINT STRING$(Length%, “#”)
     Length% = Length% - 2
     IF Length% < 0 THEN Length% = 0
NEXT I%
SLEEP
END
```

CHAPTER V, EXERCISE #4

Exercise #4: (harder). Make a program that is able to calculate all combinations of numbers within a specific range and length defined by the user. So if the user tells your program to show all combinations with length 4 and range 2, the program should output 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, and that this are 16 combinations. Have the maximum range be 10 and the maximum length be 8.

```
DO
     INPUT “Lenght: “, Length%
LOOP UNTIL Length% > 0 AND Length% <= 8
DO
     INPUT “Range: “, Range%
LOOP UNTIL Range% > 0 AND Range% <= 10
CLS
Combi& = 1
AtEnd% = 0
Combin$ = STRING$(Length%, “0”)

DO
     IF AtEnd% = 0 THEN
          PRINT Combin$
     ELSE
          PRINT “These were”; Combi&; “combinations”
          EXIT DO
     END IF

     TryPos% = Length%
     DO
          Value% = VAL(MID$(Combin$, TryPos%, 1)) + 1
          IF Value% < Range% THEN
               MID$(Combin$, TryPos%, 1) = LTRIM$(STR$(Value%))
               Combi& = Combi& + 1
               EXIT DO
          ELSE
               MID$(Combin$, TryPos%, 1) = “0”
               TryPos% = TryPos% - 1
               IF TryPos% = 0 THEN
                    AtEnd% = -1
                    EXIT DO
               END IF
          END IF
     LOOP
LOOP
```

All answers were created in three-quarter of an hour, without pre-knowledge. There may be bugs in the code. Please report them.