QB-Tutorial

By Neo Deus Ex Machina
http://www.blindcoding.cjb.net

Beginners & Newbies Edition

WHY USE QB? (I)

Since QB is very versatile, it is a must know for today's programmers. Almost everything can be done with QB. Although today's fashion isn't DOS anymore, it can be quite useful to know something about DOS and about QB, which is a DOS-oriented programming language.

QB certainly is the easiest programming language of all to learn. Also, when knowing this language, other programming languages are very easy learn, like Visual Basic, or even C or Java. Also, beginning with a DOS-language, opens new ways and knowledge when stepping to Windows-based languages. Because the most you can use in DOS can also be used in Windows.

Although QB isn't too fast, a lot can be done with it, easily.

THIS TUTORIAL

In this tutorial, specific ways of text-decoration are used to keep things clear. I will explain them now.

This is normal text decoration… ;-)
This is a remark
```
This is code-decoration
```
This is an exclamation
A note

I think that was everything. Although still a note: not all examples in this tutorial are tested. So there might be some bugs in them.

QB: Some Basics (II)

Variables

Variables are temporary storage places of different kinds of values. Variables have a unique name. There are different types of variables which can all contain different values. (For the types, see section 'Data Types'). To store something in a variable, use the equals-character (the =). To get the value that is stored in the variable, just type its name.

If you still don't understand this explanation, try this: a variable can be seen as a drawer in e.g. a closet. You can put things in the drawer, and you can look into a drawer to see what's in it, you can remove the contents of the drawer, and you can remove the drawer completely.

Data Types

There are 6 data types available in QB, which should be more than enough. ☺. They are all explained here. Well, here they come:

Integer is the first and most important data type. This data type is the most used data type of all. You can store only non-decimal numbers in an integer. Since an integer contains 16 bits, you can only store values from –32768 up to 32767.

Long integer is the second. Long integers are not used that often, since mostly integers work fine. Long integers are actually the same as normal integers, so only non-decimal numbers. However, a long integer contains 32 bits, allowing you to store values from –2147483648 to 2147483647.

Single, which can contain decimal numbers. Singles are 32 bit and you can store values from around $9^{e}40$ to 9^{e}-39.

Double, which can also contain decimal numbers. The only difference is that doubles are 64 bits large, thus providing space for values from around $9^{e}305$ to 9^{e}-310.

String. The string is a special type. It can contain characters like 'A' or '*f*'. Strings are always notated with double quotation marks around them. (“). The length of the character-list can range up to 64k.

String * n%, which is a string with a set length. Viz.: n characters can fit in the string.

That were them all. Each data type also has its own postfix-sign. I.e. that if you call a variable for instance 'number%', the variable is called 'number' and declared as an integer. If variables don't have a post-fix sign, they are singles, unless you said them not to be ;-). Here are all postfixes:

Integer: **%**
Long: **&**
Single: **!**
Double: **#**
String: **$**

Remarks In Code

You can also put remarks in your code, which can make your code more clear to yourself and other people reading your program. To make a remark in code, use a single quotation mark (') followed by the remark. You can also use the command called 'Rem' which actually does nothing so it's suitable for remarking (Rem is an abbreviation of remark).

QB: First Coding (III)

Your First Program

Start up QB, by double-clicking on QB.EXE. Then type the following code:

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

Then run the program by pressing the Shift key and the F5 key simultaneously. And… There you see your first program. Although the program will be ended immediately after you pressed F5, the program still gives some output. To get used to the commands used in a program, I always show a table with all commands you haven‘t had yet.

`PRINT [var]`	Print shows something on the screen. The command itself accepts everything from strings to numerical values.
`END`	Stops your program (and returns to the QB-IDE if running from the IDE).

You don‘t have to write the commands all in capital, the IDE does this automatically for you.

You can always use the keycombination [Ctrl]+[Break] to immediately stop the program.

Locating And Colouring Some Text

Type this example in the QB IDE:

```
COLOR 7
PRINT “This is grey text!”
COLOR 4
PRINT “This is red text! :)”
COLOR 15
PRINT “White text!”
COLOR 0
PRINT “You can‘t see black text though! :)“
END
```

Run the program. If you already knew what Print did, you’ll probably now know what Color does.

`COLOR fground% [, bground%]`	Sets the current foreground color to fground% and the background color to bground% (bground% is optional, thus you don‘t have to specify one). (Note the integer-postfix, so number must be an integer). For now, we only use 16 colors: 0 = Black 1 = Blue 2 = Green 3 = Cyan 4 = Red 5 = Magenta 6 = Brown 7 = Gray 8 = Dark gray 9 = Light blue 10 = Light green 11 = Light cyan 12 = Light red 13 = Light magenta 14 = Yellow 15 = White You should know these values and colors by heart!

This isn‘t the end. You’ll now watch how you can position some text on the screen:

```
CLS
LOCATE 1, 1
PRINT “Upper left”
LOCATE 1, 69
PRINT “Upper right”
LOCATE 13, 30
PRINT “Somewhere in the middle”
LOCATE 25, 69
PRINT “Lower right”;
END
```

Using locate, we can though position text on the screen. With positioning, it might be handy to know what the numbers behind Locate mean. Well, the first is the y-coordinate, the second is the x-coordinate. The y-coordinates range from 1 to 25 (1 is the first line, 25 the last). The x-coordinates range from 1 to 80 (1 is the most left and 80 the most right).

`CLS`	Clears the screen.
`LOCATE y-cord%, x-cord%`	Sets the cursor position to the coordinates (y-cord, x-cord). The next Print-command will put the text on those coordinates.

Of course, Locate can also be used in conjunction with Color to create colored and positioned text. *Note that Locate only applies to the next Print-command encountered, Color applies to all Print-commands until another color is set using another Color-command.*

Did you notice the ‘;’ behind the last print command? I hope so… A ‘;’ has been put at the end of the print command because the last two lines of the screen (y-cords 24 and 25) are a little weird. Normally, when you put text there, the entire screen shifts 1 line up, thus removing everything on the first line. The ‘;’ at the end of a print command, prevents the screen from shifting up.

The First Timers

Of course QB has the possibility to setting the commands on time. So here we’ll explain two important function of the command called ‘Sleep’. Sleep has the ability to wait a number of seconds before proceeding with the next command. The second function of Sleep is wait, till the user presses a key. They are both demonstrated in the following program:

```
CLS
PRINT “Please wait 3 seconds…”
SLEEP 3
PRINT “Now press a key to end the program”
SLEEP
END
```

`SLEEP [number%]`	If you specify a number behind sleep, the Sleep command waits that many seconds before proceeding. If you don‘t specify a number, Sleep waits until a keypress before proceeding.

Pressing a Control-key during a Sleep-command (with a value specified), will cause the Sleep command to be skipped. So if you say ‘Sleep 1000’ and someone presses the Control-key, the computer immediately proceeds with the next command.

Using Variables In Your Code

As said before, variables can be used in your code for dynamic numbers or characters. First look at this example:

```
CLS
DIM MyInteger AS INTEGER
DIM MyString AS STRING
MyInteger = 7
MyString = “My name is Neo”
PRINT MyString
PRINT “And my lucky number is:”; MyInteger
END
```

As you could see in this example, first, 2 variables are defined. MyInteger is defined as an integer and MyString is defined as a string. Then, the value 7 is put into MyInteger (assigned to…), and the text “My name is Neo” is put intro MyString. Then, MyString is printed to the screen. Finally, the text “And my lucky number is:” is printed on the screen with at end of the text a 7.

Do not confuse PRINT “MyString” with PRINT MyString. (The quotation marks are the difference). The reason is very simple: the first print prints the text “MyString” literally to the screen, and the second prints the contents of the MyString-variable to the screen.

`DIM varname AS vartype`

This commands defines a variable called ‘varname’ as ‘vartype’. Varname may not begin with a number and may not contain extended characters like é or Æ.

```
PRINT [text]; [text]
PRINT [text], [text]
```

We already spoke about Print before, but I think the ‘;’ needs extra attention. The thing is, Print is not only able to print 1 value to the screen, it can also print different values to the screen. To gain this effect, use a ; to separate different values. So if you do this: PRINT “Hello”; “ my name is Neo” you’ll get this on the screen: Hello my name is Neo. So the ‘;’ itself doesn‘t contain spaces, it just pastes the following value to the last value. This isn‘t it, because you can also do this: PRINT “Hello”, “my name is Neo” (with a comma instead of a semicolon). The output is different now, because the comma doesn‘t paste values together with nothing between them, but it pastes values together with some spaces between them, *keep this in mind!*

Asking The User For Values

QB is also able to let the user enter some values which you may find handy. Here’s an example:

```
CLS
DIM number AS INTEGER, answer AS INTEGER, yourname AS STRING
INPUT “Now enter a number:”, number
INPUT “Now your name:”, yourname
```

```
PRINT “So “; yourname; “ you wanted to know what is”; number; “ times 5?”
answer = number * 5
PRINT “Well, it is:”; answer
END
```

You see, when you run this program, you are asked for a number and your name. It shows some text and the correct answer of the number you entered times 5. (the asterisk means ‘times’).

```
INPUT [text$,] variable
INPUT [text$;] variable
```

First, it shows the text stored in text$, then asks the user to enter a value for variable. If you use a comma or a semicolon as separation-mark you’ll get a question mark or no question mark at the end. (A comma gives none, a semicolon gives one at the end of the text).

Note: you don‘t have to put double quotation marks when entering a string value. If your program asks you to.

Chapter Program

You probably don‘t know yet, but at the end of each chapter, there is an example program and some exercises about everything you know so far. You at least should have made all exercises before proceeding to the next chapter.

```
CLS
DIM favoritecolor AS INTEGER

LOCATE 10, 10
PRINT “Please enter your favorite color-number beneath:”
LOCATE 11, 10
INPUT “Yes here:”, favoritecolor
COLOR favoritecolor
PRINT “So your favorite colornumber is:”; favoritecolor
PRINT “Now press a key to proceed”
SLEEP
PRINT “Quitting this program… Please wait 5 seconds”
SLEEP 5
END
```

Now for the exercises…

Exercise #1: Show all colors on screen by showing the text ‘this is …… text’ with on the …’s the color’s name.

Exercise #2: Same as Exercise #1 but now show it exactly at the most right of the screen.

Exercise #3: Ask the user for his ASL then show the statistics in an emptied screen.

Exercise #4: Make a kind of interview with the user with about 10 questions. At the end, show a report about the user’s answers. (E.g. if the user said his name is ‘Harold’ in the report it must say ‘His name is Harold’).

Modifiers & Structure (IV)

Datatype Modifiers

A string without the possibility to change it isn‘t a string. And the language in which that isn‘t possible, sucks. Here’s an example program showing some of the datatype-modifiers:

```
CLS
DIM yourname AS STRING
INPUT “Please enter your name:”, yourname
PRINT “Your name in upper case: “; UCASE$(yourname)
PRINT “In lower case: “; LCASE$(yourname)
PRINT “Length of your name:”; LEN(yourname)
PRINT “First letter: “; LEFT$(yourname, 1)
PRINT “Last letter: “; RIGHT$(yourname, 1)
PRINT “Second letter: “; MID$(yourname, 2, 1)
INPUT “Please enter a lot of spaces, some letters and again a lot
of spaces:”, strangestring$
PRINT “Without the first spaces: “; LTRIM$(strangestring$)
PRINT “Without the last spaces: “; RTRIM$(strangestring$)
PRINT “Just a couple of spaces: “; SPACE$(10)
INPUT “Please enter a number:”, number%
PRINT “Stringed number:”; STR$(number%)
PRINT “Numericaled string: “; VAL(STR$(number%))
INPUT “Now enter 1 character:”, char$
PRINT “ASCII-code of char:”; ASC(char$)
INPUT “Enter a value between 33 and 255:”, value%
PRINT “Corresponding ASCII-char: “; CHR$(value%)
END
```

That were most of them indeed. Now, let’s explain.

`UCASE$(astring$)`	This function converts astring$ to upper case and returns it.
`LCASE$(astring$)`	This function returns a string equivalent to astring$ in lower case.
`LEN(astring$)`	This function returns the length of astring$.
`LEFT$(astring$, pos%)`	This function returns the first pos% characters of astring$.
`RIGHT$(astring$, pos%)`	This function returns the last pos% characters of astring$.
`MID$(astring$, pos%, len%)`	This function returns a string which is a part of astring$. The string it returns starts at pos% and has a length of len %. So if you do PRINT MID$(“HELLO”, 2, 2) on the screen it’ll say EL.
`LTRIM$(astring$)`	Removes all spaces in front of a string, so if you got astring$ = “ hello” then Ltrim$ will return “hello”.
`RTRIM$(astring$)`	Removes all spaces at the end of a string. See Ltrim$
`SPACE$(number%)`	Returns a string with number% spaces in it.
`STR$(number%)`	Returns the string-equivalent of a value. So if you’ve got a = 56 and then do mystring$ = STR$(a) you’ll get mystring$ = “ 56”
`VAL(astring$)`	Converts a numerical string to its corresponding value.

	This is the inverse function of Str$.
`ASC(char$)`	Returns the ASCII-code of char$.
`CHR$(asciicode%)`	Returns the character with ASCII-code asciicode%.

I understand if you can‘t understand it all at once, but that is no problem. The only thing you haven‘t had yet are Functions, Statements, Expressions and Metacommands. I’ll explain them here (except the metacommands).

If a command is a statement, that command is a master-type. What do I mean with that? Well, for example PRINT is a statement. Statements are commands which do something, but do not return anything. They just do something. Like PRINT, it prints something on the screen, and it doesn‘t do more.

Well, the thing is, that there are Functions too. Functions always return a value or a string and can never stand-alone. Just try to type CHR$(139) in the IDE and try to run. You’ll get an error! Why? Simply because CHR$ returns a value, but the QB IDE can‘t do anything with it, does the IDE has to show it on the screen? Or does it have to put it in a variable? That’s why functions never are standalone, you always have to specify what the QB IDE has to do with the return value. You can say: PRINT CHR$(139) or MyString$ = CHR$(139). Because now the IDE knows what it has to do with it. In the first case, it prints the returns value of Chr$(139) to the screen, in the latter, it stores the return value of Chr$(139) in MyString$.

QB also knows things that are called ‘expressions’. Expressions are some kind of formulas like 3 + 1. You have often used expressions already, although you probably didn‘t know that. Expressions can also contain functions like this one: 3 + SQR(5). The SQR function simply calculates the Square Root of, in this case, 5. Just like functions, expressions are not standalone. This is very easy to explain. For instance, if you type 3 + 3 in the IDE then try to run it, you’ll get errors, why? Because when the compiler sees the line ‘3+3’ you just typed, it thinks ‘hrrmmm 3+3 and then? 3+3 so what? What do I have to do with 3+3?’ You can for instance specify that the answer of 3+3 must be printed to the screen, or that it must be stored in a variable. By doing this:

```
PRINT 3 + 3
Answer% = 3 + 3
```

Also, functions and expression can be nested, the statements cannot. Simply look at the following example, it looks kinda complex, but it is actually the ABC-formula (Maths!! Remember?).

```
PRINT “This is the ABC-Formula Calculator”
PRINT “You must enter values with the corresponding”
PRINT “formula: y = a * x ^ 2 + b * x + c”
INPUT “Please enter a:”, a!
INPUT “Please enter b:”, b!
INPUT “Please enter c:”, c!
PRINT
D# = b! ^ 2 – 4 * a! * c!
PRINT “Discriminant =”; D#
X1 = (-b! – SQR(D#)) / (2 * a!)
X2 = (-b! + SQR(D#)) / (2 * a!)
PRINT “First zero-point at x =”; X1
PRINT “Second zero-point at x =”; X2
END
```

I hope you understand it… ;-).

Structure I: Choices

QB is also able to make choices, by a special command called ‘IF’. You must specify an expression (which must return a boolean) behind the IF, then a ‘THEN’ then type what is going to happen if your expression is true. Perhaps it’ll be more clear if I show you an example:

```
PRINT “Please enter a number”
INPUT “Number:”, number%
IF number% = 0 THEN PRINT “You entered a zero”
IF number% <> 0 THEN PRINT “You didn‘t enter a zero”
```

When you read this program, it should have been cleared up by now, because you now know that the first If checks if number% = 0 and if so, it prints ‘you entered a zero’ on the screen. The second checks for number% ≠ 0 and if so, it prints ‘you didn‘t enter a zero’ on the screen.

In QB, the character ≠ doesn‘t exist. Instead of that, you should use <> for ‘not equal to’.

Actually, the first example was inefficient, because 2 if commands were needed. I could extend your knowledge now by showing you this example:

```
PRINT “Please enter a number”
INPUT “Number:”, number%
IF number% = 0 THEN PRINT “ZERO!” ELSE PRINT “No Zero! :(”
```

You’ll probably already understand what this program does, but let’s explain anyhow. The If command now checks if number% is equal to 0, if so, it prints ‘zero!’ on the screen, if not it prints ‘no zero!’ on the screen.

This If-structure can be more extended by starting a second line. This is very handy, because then you can execute more commands when an expression is true. Watch this:

```
PRINT “Please enter a number”
INPUT “Number:”, number%
IF number% = 0 THEN
   PRINT “Division by zero isn‘t allowed.”
   PRINT “Please try again”
   END
ELSE
   PRINT “This entry will not cause an error”
   PRINT “Because there is no division by zero”
   PRINT “The answer is:”; 1 / number%
END IF
```

I think this example should be clear enough. For the ones who don‘t understand, this program will calculate 1/number% for a number entered by the user. If the user enters a 0, the part between the If and the Else will be executed. If it isn‘t a 0, the part between the Else and the End If is exectuted.

Note: if you put the whole IF statement on one line, which is possible, you don‘t have to use an ‘End If’. This is only used for If statements which are not written on one line. The End If simply defines where the piece of code, which belongs to an If-statement, ends.

There’s actually 1 function of If left, which is ‘Elseif’. Try to understand this example:

```
PRINT “Please enter a number”
INPUT “Number:”, number%
IF number% < 0 THEN
   PRINT “The number is smaller than zero”
ELSEIF number% = 0 THEN
   PRINT “The number is zero”
ELSE
   PRINT “The number is larger than zero”
END IF
END
```

As you can see here, the Elseif-part of an If-statement simply is some kind of another If statement. The only difference between writing it like in the example and writing it separated (2 different If-statements) is that, when you write them separated, they can both be executed, if you write them as above, only 1 of the choices that are presented will be executed, which usually is the first If-part, which has an expression which is true.

You can add as much Elseif-parts as you want. Also, the Else part is not necessary. Also, Elseif-parts may only be used if an If-statement is not on 1 line, with Else this is possible. (See 2nd example of this chapter).

For large selections, you shouldn‘t use If, but Select, which is the following subject. (Because writing 100 Elseif’s is quite some work ;-)).

Structure II: Choices

The syntax of Select is very easy. See this example (which is the same as above):

```
PRINT “Please enter a number”
INPUT “Number:”, number%
SELECT CASE number%
   CASE 1, 2, 3
      PRINT “The number is either 1, 2 or 3”
   CASE IS > 3
      PRINT “The number is greater than 3”
   CASE IS < 0
      PRINT “The number is smaller than 0”
   CASE ELSE
      PRINT “The number is a number which I don‘t know”
END SELECT
END
```

Select not only is less typing, it’s also much clearer than If. Everyone should understand what this program does.

`SELECT CASE varname`	Defines the beginning of a Select-part. Varname is the variable which has to be checked using this Select structure.

`CASE option [, option, ……]`
`CASE IS expression`
`CASE ELSE`
	Case defines a code part for if Varname = one of the options specified. You can also use an expression in conjunction with IS, because… try this without IS: varname is smaller than 0. (With options). Then you’ll

have to enter every value below 0. It is much handier if you say CASE IS < 0 instead of enumerating all possible values below 0. The code below Case Else will only be executed if no other specified cases are true. Also, the code that will be executed when a Case is true, will be everything until a next Case or an End Select is reached.

`END SELECT` Denotes the end of a select structure.

Structure III: Jumping

Jumping is not too often used, but is worth talking about. (Simply because newbies and beginners use jumping a lot ;-)). Jumping can be achieved by 2 commands, Goto and Gosub. I'll also explain them in that order. So goto first:

```
Again:
PRINT "I'll calculate 1/n for you"
PRINT "(type a 0 to end)"
INPUT "Please enter n:", n%
IF n% = 0 THEN GOTO nowend
Answer# = 1# / n%
PRINT "The answer is:"; Answer#
PRINT
PRINT "Do you want to enter another n?"
TypeAgain:
INPUT "(j/n):", yesorno$
Yesorno$ = UCASE$(yesorno$)    'convert to uppercase
IF yesorno$ = "J" THEN
   GOTO Again
ELSEIF yesorno$ = "N" THEN
   GOTO nowend
ELSE          'user typed no N or J
   GOTO TypeAgain
END IF

nowend:
END
```

As you could see in this very exaggerated program, Goto is used to jump back in your code (like GOTO Again does), or forward (like GOTO nowend). Goto must be used in conjunction with a label. You must specify a label somewhere in your code, by typing your label's name followed by a colon ":". Then you can use Goto yourlabel'sname to jump to your label. From there, the program starts running as normal.

`Label:` Defines a label with name 'label'. You can fill your own name in for 'label', like done in the example.

`GOTO label` Jumps to the label called 'label'. From there, the program will continue.

If you refer to a label (using Goto) that doesn't exist, you'll get an error.

In some cases, Goto can be handy. However, overuse of the command will result in what is called 'spaghetti-code'. The code will then be that complex, that even the best programmer can't

understand the code, which is due to the fact that when using Goto, the code can be run on numerous ways. So use Goto with caution.

That's why they invented 'Gosub'. Which will, in some cases, work more efficient than Goto. By declaring a module-level sub, this works almost the same as a procedure or function (Sub or Function). The only difference between Gosub and Goto is that Gosub requires a 'Return' statement. As you'd already seen in the example above, Goto doesn't require any. Well, let's just show you an example:

```
CLS
PRINT "I can calculate contents"
PRINT "Please enter which contents you want to calculate"
PRINT "(cube, cylinder, ball)"
Again:
INPUT "Type it here:", choice$
Choice$ = LCASE$(LTRIM$(RTRIM$(choice$)))
Cont% = 0
SELECT CASE choice$
     CASE "cube"
          INPUT "Please enter l:", l%
     CASE "cylinder"
          INPUT "Please enter r:", r%
          INPUT "Please enter h:", h%
     CASE "ball"
          INPUT "Please enter r:", r%
     CASE ELSE
          GOTO Again
END SELECT
IF choice$ = "cube" THEN
     GOSUB calccubecont
ELSEIF choice$ = "cylinder" THEN
     GOSUB calccylindercont
ELSEIF choice$ = "ball" THEN
     GOSUB calcballcont
END IF
PRINT " Contents:"; Cont%
END

Calccubecont:
Cont% = l% ^ 3
RETURN

Calccylindercont:
Cont% = 3.1415! * (r% ^ 2) * h%
RETURN

Calcballcont:
Cont% = (4 / 3) * 3.1415! * (r% ^ 3)
RETURN
```

I think it is clear by now, but let's explain anyhow. The Gosub command on line 21, refers to a label called 'calccubecont'. In the previous section about Goto, I've already explained what a label can do. The program then jumps to that label, running everything what is said below. Until the

program arrives at a ‘Return’. Which is used to show the program the piece of code to be run is over and the program jumps back to the Gosub command it came from, but now ignoring it and thus, proceeding with the code after the Gosub command.

`GOSUB label`	Works almost the same as Goto. Have the program jump to the specified label.
`RETURN`	Gosub requires this command, specifying the end of the ‘label-code’. When reaching this command, the program jumps back to the last Gosub command it came from.

When either overusing the Gosub command, or nesting them, this will result in a memory error. So try to avoid it.

Nesting is consequently putting commands of the same purpose in eachother. This is e.g. possible with If-statements (putting an If inside an If, which also is inside an If). It can also be done with Gosub or Select. Later on, you’ll learn that it is also possible with Do, For and While.

Chapter Program

Ready? I hope so.

```
Again:
CLS
COLOR 15, 1
PRINT “==================================”
PRINT “   ===========      ===========”
PRINT “      === Palindromotron ===”
PRINT “   ===========      ===========”
PRINT “==================================”
COLOR 7, 0
PRINT “This program is capable of reversing”
PRINT “words, sentences or entire texts.”

DIM TheString AS STRING, EndString AS STRING
INPUT “Please enter some:”, TheString
TheString = LTRIM$(RTRIM$(TheString))
IF LEN(TheString) = 0 THEN GOTO Again

GOSUB reversestring
PRINT “This is it, reversed: ”; EndString
PRINT
EnterAgain:
PRINT “Do you want to enter some text again? (y/n)”
INPUT “(y/n):”, choice$
Choice$ = LTRIM$(RTRIM$(UCASE$(choice$)))
IF Choice$ = “Y” THEN GOTO Again
IF Choice$ = “N” THEN END
‘the program will only get here if Choice$ ≠ “Y” AND Choice$ ≠ “N”
GOTO EnterAgain

Reversestring:
‘this function can actually be made much better using FOR or WHILE
‘but since you haven‘t had them yet, I do it with GOTO
```

```
NextChar:
CurrentPos = LEN(TheString)
EndString = EndString + MID$(TheString, CurrentPos, 1)
CurrentPos = CurrentPos – 1   ‘decrease CurrentPos with 1
IF CurrentPos <> 0 THEN GOTO NextChar
RETURN
```

EXERCISES

Exercise #1: Make a kind of quiz. Show about 20 questions and at the end how many you had guessed wrong.

Exercise #2: Make a program that counts the number of e or E in an entered string, and shows that number.

Exercise #3: Make a program which asks the user to enter as much values the user would like to, then calculates the average of the numbers.

Exercise #4 (HARD!): Make a text adventure. If you don‘t know what a text adventure game is, try downloading ‘Neptune Caverns’ at www.nemesisqb.com.

Loops & Boolean Operators (V)

Boolean Operators

Boolean Operators are used to paste two or more expressions together. E.g. you made a database program about your classmates. You search through the database for someone who's called 'BlueKeyboard' and lives in a town called 'aetherFox' and is 15 years old. You could do this like this:

```
IF name$ = "BlueKeyboard" THEN
     IF city$ = "aetherFox" THEN
          IF age% = 15 THEN
               'yeah!!! You found him!
          END IF
     END IF
END IF
```

This code looks clean. But can you imagine how much you have to type when you have more criteria? Look at this one:

```
IF (name$ = "BlueKeyboard") AND (city$ = "aetherFox") AND (age% =
15) THEN
     'found!!!!
END IF
```

If you pronounce the code just like it says, it is already clear. In this code, 'AND' is a boolean operator, which, in this case, pastes two criteria together. Here's the rest:

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

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

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

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

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

	True	*False*
NOT	False	True

These schedules are called 'TF-tables' which stands for 'True-False Tables'. In these schedules you can see what an operator returns when evaluating the criteria. As you already know, an If-statement is only run if the criteria is TRUE. So if the criteria is '5 = 5' then this will be TRUE because in this case, five will always be five. However, when you use 2 or more criteria, you'll need to specify the link between the criteria. E.g. does every criteria has to be true? Then use AND. Here's some code to get used to it:

```
A = 1
B = 2
C = 3
D = 4
IF (A = 1) AND (B = 2) THEN
'this will only be run if A is equal to 1 AND B is equal to 2.
END IF
```

```
IF (A = 2) OR (C = 3) THEN
‘this code will only be run if one of both criteria is true, or
‘when both are true. So this code will be run when:
‘A = 2 and C <> 3
‘A <> 2 and C = 3
‘A = 2 and  C = 3
‘so or is very flexible.
END IF

IF (C = 3) XOR (D = 3) THEN
‘this code will only be run if one of both criteria is true.
‘if both are true, this is not run (also when both are false).
‘So this code will be run when:
‘C = 3 and D <> 3
‘C <> 3 and D = 3
END IF

IF (D = 4) EQV (A = 1) THEN
‘this code will only be run if both are true, or both are false.
‘So it will be run when:
‘D = 4 and A = 1
‘D <> 4 and A <> 1
‘I seldom use EQV though
END IF

IF (B = 3) IMP (A = 1) THEN
‘this code will be run when both are true, when both are false
‘or when only the first one is true (if the second is true and
‘the first is not, this code is not run).
‘So when:
‘B = 3 and A = 1
‘B <> 3 and A <> 1
‘B = 3 and A <> 1
‘Never use it often though
END IF
```

If you’d studied the TF-Schedules well, you would have noticed one thing. About the ‘NOT’ operator. The difference is that, AND, OR, XOR, EQV and IMP are binary-operators, and that NOT is an unary-operator. What does this mean? Well it’s actually very simple. The binary operators link 2 criteria together, while unary operators link only 1. So NOT only links 1. But how? Simple, NOT turns around the value of all inner values. Example:

```
IF NOT(A = 5) THEN
‘this code is only run if A is not equal to 5. If A = 5 then the
‘expression between the brackets is true, because A = 5. Then not
‘turns around that value, so it is converted to false. Also, when
‘A = 4, the inner expression is false, because A = 5 is false
‘because A = 4, so not converts false to true, and the if-clausule
‘is run.
END IF

IF NOT(A = 1 AND B = 2) THEN
‘this code is run if the inner expression is false (not converts
```

```
‘false to true). In the text above about AND you can find out when
‘the inner expression will be false.
END IF
```

Till now, I kept it easy. But you can also imagine that sometimes, people need very complex structures. When using more than 1 binary operator, you should also apply brackets to your code so the QB IDE knows which binary operator to evaluate first. Here’s a very complex example:

```
IF (((score = 200) AND (level = 5)) OR ((deaths = 10) EQV (units >
200))) AND (NOT(U = 9 IMP kills < 1) XOR (skill <> 5)) THEN
‘find out yourself when this code will be run!! :-D ;*)
END IF
```

Loops I: For … Next

There are 5 ways to incorporate loops inside your program, of which I learnt you 2 of them (viz. Goto and Gosub). This sub-chapter is about the For loop. For is actually a counter-loop. You specify a start value and an end value. And the internal variable keeps increasing until it passed the end value. Then a For-loop is exited. Try this very clear example:

```
FOR internalvar = 1 TO 100 STEP 1
     PRINT internalvar
NEXT internalvar
```

As you can see when you run this program, the numbers 1 up to and including 100 are printed on the screen. How’s that possible? Well, first you say that internalvar = 1, it has to go too 100 with steps of +1 in size. Thus, the for equals internalvar to 1, runs all the code beneath, until it encounters a ‘Next’ command. Here it says ‘Next internalvar’ so the program jumps back to the ‘For internalvar …etc….’ and increases internalvar by step, which here is 1. So the second time the code will be run, internalvar will equal 2. Then 3, then 4, 5, 6, 7, etc… up to and including 100. Then the program sees the specified limit has been reached and now skips the ‘Next’ command when it arrives there.

The for-routine is very flexible and can almost be used for everything what is in need of a counter. Take a look at this example:

```
PRINT “Please enter some text”
INPUT “Here:”, sometext$

FOR counter = 1 TO LEN(sometext$)
     PRINT MID$(sometext$, counter, 1)
NEXT counter
‘the loop will run from 1 to the length of the text you entered.
‘then it prints the counterth character of the text on a line.
```

It doesn‘t matter if you can‘t yet understand this. The only thing that matters is that you understand what For does and how and when to use it.

The step-factor is optional, but when left away, the (virtual) step-factor will always be 1.

Loops II: Do … Loop

The second, but just as important, loop is the Do-Loop routine. On the contrary to For-Next, Do-Loop is not a counter. Do-Loop can be 2 different kind of things. Viz. an endless loop (also:

‘eternal loop’), or an expression-waiter (loops around until an expression is true or while an expression is true). Here’s an example about the eternal loop:

```
DO
     PRINT “Always the same text!!! :*)”
LOOP
```

Press [Ctrl] + [Break] simultaneously to immediately stop the program and return to the IDE.

As you can see when running this program, the commands between the Do-Loop are run for ever and therefore the program will never stop. How it actually works is quite simple: the program encounters a ‘Do’ and remembers where it was, runs some commands, then encounters a ‘Loop’ and the program will jump back to the last ‘Do’ it encountered (actually it is the do where the loop belongs to).

Here are 2 examples about the expression-loop:

```
DIM I AS INTEGER
I = 0
DO
     I = I + 1
     PRINT I
LOOP UNTIL (I = 10000)

PRINT “First example ended.”

I = 0
DO WHILE I < 10000
     I = I + 1
     PRINT I
LOOP

PRINT “Loops ended.”
END
```

You can see the expression can be added to a Do-Loop routine by adding an Until or a While. Both need an expression, or more linked by using Boolean Operators. As you could have already known (by pronouncing the code), a Do-Loop routine with While, will run the code while some kind of expression is true, and it won‘t be run anymore if that expression is false. The Do-Loop routine with Until will run while some kind of expression is not true, and will exit the loop when the expression is true. (Pronounce: Loop the loop while my score is smaller than 100, or Loop the loop until my score is greater than 100, for example).

```
I = 1
PRINT “From the first loop:”
DO
     PRINT I
LOOP UNTIL I = 1

PRINT CHR$(13); “From the second loop:”
DO UNTIL I = 1
     PRINT I
LOOP
```

The example above was dealing with the difference between Do Until and Loop Until (what will be explained now is the same for Do While and Loop While). As you could have noticed when running the program, from the first loop it says '1' and from the second loop it says nothing. This phenomenon can easily be explained, let's just start at the first loop (I show you what the program 'thinks').

Line 1: aah, a new variable called I. Store value 1 in I.
Line 2: Print "From the first loop" to the screen.
Line 3: aah, the start of a Do-Loop.
Line 4: Print I to the screen.
Line 5: hrrmm, 'Loop until I = 1', well, I is equal to 1 so let's exit the loop.
Line 7: Print an enter and the text "From the second loop" to the screen.
Line 8: Start of a Do-Loop, and since the loop must be exited when I equals 1, and since still I equals 1, I won't run the loop and will jump immediately to the Loop the Do belongs to.
Line 10: I jumped from Line 8 to Line 10.

It should be clear now, if not, play some with the Do-Loop :*). It will be clear to you in no time. And… if you still don't understand after playing with it, here's some very short explanation:

When the program encounters a Do-Loop Until, the program first runs the code, and at the 'Loop Until' it checks if the expression is true, if so, exit the loop, else, loop again. But when the program encounters a Do Until-Loop, it first checks if the expression is true, if so, exit the loop, else, loop the loop, and when it arrived at the Loop, it jumps back to the Do Until and checks again, if the expression is true now, the program will jump forward to the Loop it came from and goes on from there.

Loops III: While … Wend

The 3rd and the most unimportant loop, is the While-Wend. This loop actually does exactly the same as Do While-Loop, so consider using that one instead. However, in some code While-Wend is often used, and is therefore worthy to be explained here. As I already told, While-Wend is similar to Do While-Loop. An expression should be specified after the While, and the loop will be run if the specified expression is true. Example:

```
Linescounted = 0
WHILE Linescounted < 100
     PRINT "This is line"; Linescounted + 1
     Linescounted = Linescounted + 1
WEND
```

This example doesn't need much explanation. Just pronounce: 'While the lines I counted is beneath the 100, do this'.

I think it's now time for an overall-view.

```
DO [WHILE/UNTIL expression]
…
LOOP [WHILE/UNTIL expression]
```

Runs commands between the Do and the Loop until or while an expression is true. When no Whiles or Untils are used, the loop will run forever.

```
FOR counter = start TO end [STEP factor]
…
NEXT counter
```

Runs commands between the For and the Next while a counter is running. The 'loop' will be exited if the counter reaches the specified end-value. The counter starts at start and ends at end, and is increased every

loop by factor. This ‘loop’ is very handy, for the counter is a variable which you can use in your code.

```
WHILE [expression]
…
WEND
```

Handy sometimes, this loop runs commands between the While and the Wend while an expression is true. The loop won‘t be run (anymore) if the expression is false.

Concerning Do-Loop, you can only specify 1 expression per routine, so you can‘t make a Do While-Loop While or a Do Until-Loop While (for example).

Fast exit command

However, your loops don‘t have to run until some expressions are true or false, they can also be stopped by another command, viz. ‘Exit’. Behind the Exit-command, you have to write what has to be exited, so you could write ‘Exit Do’, to exit a Do-Loop. Here’s what you have to know for now:

Exit Do – Immediately exit a Do-Loop
Exit For – Immediately exit a For-Next

Example:

```
I = 0
DO
     I = I + 1
     IF I = 10000 THEN EXIT DO
LOOP
PRINT “Loop exited at I =”; I

I = 0
FOR I = 0 TO 10000 STEP 1
     IF I = 5000 THEN EXIT FOR
NEXT I
PRINT “For exited at I =”; I
END
```

I think this is very easy to understand.

`EXIT DO` Immediately exits the first specified loop. The loop of that kind is exited if this
`EXIT FOR` command was in one of them.

Chapter Program

```
DO
     CLS
     PRINT “Prime number calculator!”
     PRINT “By Mr. X”
     PRINT
     PRINT “This program calculates all prime numbers up to”
     PRINT “a number n (N > 10)”
     N% = 0
```

```
     WHILE N% <= 10
          INPUT “Please enter n:”, N%
     WEND

     PRINT
     PRINT “1”
     PRINT “2”
     FOR primec = 3 TO N% STEP 2   ‘step 2 because only odd nrs.
          Dividable = 0
          FOR primed = 2 TO primec – 1 STEP 1
               IF primec / primed = INT(primec / primed) THEN
                    Dividable = 1
                    EXIT FOR
               END IF
          NEXT primed
          IF Dividable = 0 THEN PRINT primec
     NEXT primec
     PRINT
     PRINT “That were all.”
     PRINT “Want to see more?”
     K$ = “a”
     DO
          INPUT “(yes or no):”, K$
     LOOP UNTIL LCASE$(K$) = “yes” OR LCASE$(K$) = “no”
     IF LCASE$(K$) = “no” THEN EXIT DO
LOOP
```

EXERCISES

Exercise #1: Make a program that can calculate the facultaty of an inserted number. If you don‘t know what a facultaty is, e.g. the facultaty of 5 is 1 * 2 * 3 * 4 * 5 = 120. The facultaty of 10 is 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 = 3628800.

Exircise #2: Make a program that can calculate perfect numbers. A perfect number is a number of which the sum of all dividers is equal to that number. E.g. 28 is a perfect number, because the dividers are: 1, 2, 4, 7, 14. And the sum of the dividers is again 28!

Exercise #3: Make a program which can calculate combinations. (For readers with a Ti-83, the possible combinations can be calculated with nCr (math-prb)). The user must be able to insert the range and the length. E.g. if the user enters a 3 for range, only the numbers 1, 2 and 3 will be used for cominations. If the user enters a 4 for length, the program calculates all possibilities with length 4. So 1111 and 1321 are 2 of the possible combinations.

Exercise #4: Easy one. Make a program which can calculate:

$$\sum_{i=1}^{n} i.$$

Which means this: 1 + 2 + 3 + 4 + 5 + … + n. So calculate the sum of all positive integers up to and including n.

Catalogue

Every command, function, keyword, (simply everything) which I wrote about is in this list. Here you can find every keyword which is dealt in this part.

```
AND
AS
ASC
CASE
CHR$
CLS
COLOR
DIM
DO
DOUBLE
ELSE
ELSEIF
END
EQV
EXIT
FOR
GOSUB
GOTO
IF
IMP
INPUT
INTEGER
IS
LCASE$
LEFT$
LEN
LOCATE
LONG
LOOP
LTRIM$
MID$
NEXT
NOT
OR
PRINT
REM
RETURN
RIGHT$
RTRIM$
SELECT
SINGLE
SLEEP
SPACE$
STEP
STR$
STRING
THEN
TO
UCASE
UNTIL
VAL
WEND
WHILE (expression-waiter)
WHILE (loop-command)
XOR
```

Afterword

QuickBASIC-Tutorial Part I: Beginners & Newbies Edition
By Neo Deus Ex Machina
Member of the Blind Coding programming team
http://www.blindcoding.cjb.net
2002-17-10

Thanks to all members of the Blind Coding programming team:
AetherFox
BlueKeyboard
Mofu
Neo Deus Ex Machina

Thanks for reading my Tutorial