Introduction

A computer program is actually just a series of instructions telling the computer what to do. A computer language acts as an interpreter. The programmer writes instructions using the language, and the language tells the computer what to do. Before we get too far, let's get familiar with few convention used throughout this tutorial.

Style	Meaning
`CLS`	Capital text in `this font` is to be typed exactly as it appears -- these are the keywords.
`LET` *varname* = expression	Text in italics represents values or variables you'll have to enter.
`READ` varname [, varname, ...]	An ellipsis (...) means you can add as many elements as you wish. Here, you can put as many values after READ as you wish, separated by commas.
`IF` test `THEN` <statements> `END IF`	A <statements> means one or more lines of code was omitted for clarity's sake.

Few terms in that table that may be unfamiliar, such as *variable*. Below is another table, this one defining some important vocabulary words that are used throughout this tutorial and beyond.

Word	Definition
Statement Command	An instruction to do something. These words can be used interchangeably most of the time;
Keyword	A word that is part of the QBasic language. When you type a keyword, QBasic will automatically capitalize it for you. Keywords are used to identify commands and parts of commands. Note that you cannot make a named constant or variable with the same name as a QBasic keyword.
String	A bunch of characters. In QBasic, strings have quotation marks around them, for example, "Kuldeep".
Number	A number. This can be subdivided into Integer Long Integer, Single Floating

	Point and Double Floating
Constant	A value (string or number) that cannot change.
Variable	A named "holder" for a string or number. Variables can have a type suffix that is added to the end of the name to identify what data it holds (non-suffixed variables default as Single). Strings are $, Integers are %, Long Integers are &, Singles are !, and Doubles are #.
Operator	One of the built-in math operations. They will be described in more detail later in the tutorial. They range from arithmetic operators (+, -, etc.), relational operators (=, >, etc.), and Boolean/binary operators (AND, OR).
Comment	Does absolutely nothing. QBasic ignores comments. With comments, you can put notes in your program to remind you what this section does.
Block	Generic term for a group of lines inside a structure.
Loop	Generic term for a group of lines executed a series of times.

Your First Program

Now, its time to write your first QBasic program. This program will display "Hello, world!" on the screen.

QBasic is a DOS based program. To run it you have to open a DOS box. Click:

a. The Qbasic icon on your windows desktop.

b.A QBasic editor will be opened for you.

c. Enter the following program.

To run this Qbasic program, press ***Shift-F5*** (or choose Run|Start from the Qbasic menu).

```
  ' Hello World program
 CLS
 PRINT "Hello, world!"
  END
```

When you run this, a blank screen will appear and the two words will appear. Observe that QBasic displayed *Press any key to continue.* at the bottom of the screen. QBasic does this when a program ends. Let's understand this simple program by looking at each individual line to see what's going on.

```
  ' Hello World program
```

This line is a *comment*. All comments in QBasic begin with an ' (apostrophe) or the *keyword* REM followed by a space. So, all this line does is tell us what the heck the program is.

```
CLS - Clear Screen
```

This line is a *command*. If you know a little DOS, you probably already know this. The **CLS** command clears the screen. I used this to get rid of everything before we wrote the text.

```
PRINT "Hello, world!"
```

This is another command. As you can probably guess, **PRINT** displays text on the screen at the current cursor position. Following the PRINT keyword is a *literal constant*, the text to display. You can PRINT just about anything.

```
END
```

This line, easily enough, ends the program.

REM Command

Syntax: `{REM | '} comment`
The REM command lets you add a comment to your code. As the syntax definition shows, you can use an apostrophe (') in place of the word REM. *comment* can be anything you want
Example: `' let's Learn QBasic, its fun!`

CLS Command

Syntax: `CLS`
The simple CLS command clears everything off of the screen and puts the cursor at the top left corner of the screen. **Example:** `CLS`

PRINT Command

Syntax: `PRINT [expression {; | ,} expression {; | ,} ...] [{; | ,}]`
The PRINT command is used to put text on the screen at the current cursor position. The syntax will take some explaining. *expression* can be any string or number expression.

Examples: `PRINT`

```
PRINT " Name", "SSN"   PRINT "My name is. . . . "; myname$;
```

END Command

Syntax: `END`
The END command quits the program and returns to the QBasic editor. **Example:** `END`

Data Types

Every variable used in the program has data type. These variables are the key to making a useful program: without them, your program will run the same way ***every time it is run***. But how do you *use* variables in the first place? A variable is created the first time it is referenced in your code, such as when you first set a value to it. As stated before, there are five types of variables. Each one has its own associated suffix to identify its type. The table below describes in more detail the five data types:

Data Type	Suffix	Description
String	$	String variables are the only variables that hold text
Integer	%	Integer variables are 2 bytes long and hold integers (numbers with no fractional part).
Long Integer	&	Long Integer variables are 4 bytes long and also hold integers.
Single	!	Single-Precision variables are 2 bytes long (usually called Single) can handle numbers with a decimal point.
Double	#	Double-Precision variables are 4 bytes long (usually called Double) can also handle numbers with a decimal point.

Arithmetic Operators:

Before we go on, let's look at the first set of *operators*. These are the *arithmetic operators*. You can use them to perform the basic arithmetic functions. You use them by putting the operator between two numeric expressions: `num1 operator num2`.

Name	Symbol	Description
Addition	+	Adds two numbers together.
Subtraction	-	Subtracts the second number from the first.
Multiplication	*	Multiplies two numbers together.
Division	/	Divides the second number into the first.
Integer Division	\	Same as division, but round off the result to the lowest whole number. In other words, it gives you the answer without the fractional (remainder) part. Ex: 5 \ 2 = 2, whereas 5 / 2 = 2.5.
Modulo	MOD	Same as division, but returns the remainder found by long division. Ex: 5 MOD 2 = 2, 10 MOD 4 = 2.

Variables should be assigned a value, to use it in the program. There are two ways to do thi.

LET Command

Syntax: `[LET] variable = expression`
The LET command assigns a variable a value. *variable* is the name of any type of variable. *expression* is an expression of the same type of variable -- number or string

Examples:
```
mystring$ = "This is a test."
result% = var1% + var2%
```

INPUT Command

Syntax: `INPUT [;] [literalstring$ {; | ,}] var [, var, ...]`
INPUT lets the user input the value of a variable or variables. *literalstring$* is a literal string expression that prints a prompt -- **Examples:**

```
INPUT "Enter a number between 1 and 10:", guess%
INPUT "What's your name and phone number"; n$, p$
INPUT ; var1!
```

Condition Testing

As you can see, in the previous programs the program starts at the top of the program and works down through your lines of code. We can divert this "natural" program flow as shown in this section. Also in the previous section we used variables to keep your program from doing the same thing every time it's run. This section will teach you the core of doing different things: *conditionals*.

The major conditional, the **IF** commands, work to direct program flow. Program flow is the term for the "path" the executed statements follow. Up until now, the program flow has started at the first line and worked down to the last line. With conditional statements, you can skip one or more lines depending on a condition.

The core to any conditional is a Boolean, or true/false, value. If the value is True (any non-zero value, preferably -1), it does one thing. A value of False (0) does another thing. You can get T/F values by using the other two kinds of operators: *relational* and *logical (Boolean)/binary*.

Relational operators are tests between two number values. Basically, you can find how two numbers relate to each other. Here's a table of all of them:

Name	Symbol(s)	Description
Equal to	=	Returns true if the two values are equal, and false if not.
Not Equal to	<>, ><	Returns true if the two values are not equal, and false if they are.

Greater than	>	Returns true if the first number is greater than the second, and false if not.
Less than	<	Returns true if the first number is less than the second, and false if not.
Greater than or Equal to	>=, =>	Returns true if the first number is greater than or equal to the second, and false if not.
Less than or Equal to	<=, =<	Returns true if the first number is less than or equal to the second, and false if not.

Now that you know all about logical and relational operators, we'll put them to use. There are two main commands you can use with conditionals: **IF** and **SELECT CASE**. Both are similar, but are better suited to some things over others. Really, IF can be used wherever SELECT CASE can, but not the other way. Let's start with the more common one, IF.

IF Command (if-then-else --single-line form)

Syntax: `IF condition THEN statement [ELSE statement]`
The IF command in this form lets you execute a line depending on a condition. *condition* is a True/False value. If *condition* is True (not 0), then the *statement* following THEN is executed. If *condition* is False (0) and an ELSE clause is included, the *statement* following ELSE is executed; or else, flow continues to the next line. This command is small and good if you only need to do one thing based on a condition.
Example: `IF guess% = 5 THEN PRINT "Good job!" ELSE PRINT "You stink!"`

IF Command (If-Elseif-else -- block form)

Syntax:

```
IF condition THEN
  <statements>

[ELSEIF condition THEN
  <statements>

  ] ...
[ELSE

  <statements>
  ]
END IF
```

As you can see, this form of IF is designed for both complex situations and large chunks of code. First, the top *condition* is tested. If true, the code between it and the next ELSEIF, ELSE, or END IF is run. If it's false, the next ELSEIF clause is tested, and so on. If none of the clauses are true, the ELSE's code is run. After going through any chunk of code, flow returns to after the END IF line. The ELSE clause is optional, and you can have as many ELSEIF clauses as you

wish.

Let's see a sample program that demonstrates the use of both forms of IF. This is a number guessing game.

Example:

```
'Number Guessing Game, version 1.0
CLS
number% = 5
PRINT "Welcome to the game! Try to guess the number I'm thinking
of!" INPUT "Guess a number between 1 and 10:", guess% IF guess% <1
OR guess% > 10 THEN
  PRINT "You guessed out of range!"
ELSEIF guess% = number% THEN
  PRINT "Terrific! You guessed right on the money!"
ELSE
  PRINT "Too bad. You missed the target."
  INPUT "Do you want to know what it was (Y/N)"; see$ IF
  see$ = "Y" THEN PRINT "The answer was"; number%
END IF
END
```

Note: In large programs you might have a number of blocks inside each other. It's easy to forget the closing statements, and QBasic gives cryptic, confusing errors when you leave a block or loop open. For example, you might get a "`Block IF with no END IF`" error when in fact you forgot to close one of your loops. Even QBasic admits it.

Now, here's another way to do conditionals. This way is preferred when you are only examining the value of one variable throughout the tests.

SELECT CASE Command

Syntax:

```
SELECT CASE expression
CASE {expression1 [, expression2, ...] | IS relational_operator expression
| expression1 TO expression2}
  <statements>

[CASE (see choices above)
  <statements>


  ] ...
[CASE ELSE
  statements

  ]
END SELECT
```

This block statement looks at the value of the beginning *expression*. It checks the first CASE.

The *expression1 [, expression2, ...]* form checks to see if the value equals a certain value. The IS *relational_operator expression* form checks to see how it relates to another value. The *expression1* TO *expression2* checks to see if it is between (inclusively) two other values. If it is found to be True, the following block of code is run. If none are found True, the C ASE ELSE block (if it exisis) is run. After a block is run, the program continues after the END SELECT keyword.

SELECT CASE is recommend when checking the value of one number, and IF when using multiple variables in your tests. It's all personal preference, though. The next program shows how SELECT CASE can be used.

Example:

```
'SELECT CASE demonstration
CLS
INPUT "Please enter your marks: ", marks%
SELECT CASE marks%
CASE < 60
  PRINT "You have a D- grade."
CASE 61 TO 70
  PRINT "You have a C- grade"
CASE 71 TO 80
  PRINT "You got a B - grade."
CASE 100
  PRINT "You did very well, Excellent"
CASE ELSE
  PRINT "You failed."
END SELECT
END
```

Iteration -- Loops

Loops are the nice and easy solution if you wanted your program to do something repeatedly. All of the loop constructions in QBasic execute a block of commands repeatedly 0 or more times.

FOR/NEXT Loop Construct

Syntax:

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

The FOR/NEXT loop iterates the commands a set number of times. You assign a numeric variable as the counter. It changes value each iteration. The start and end values are the first and last values *counter* will be. You can also specify the increment, which can be positive or negative but not 0. If you omit it, it defaults to 1. The FOR/NEXT construct is ideal for blocks of code you want to run *n* times.

Example:

```
FORI=1TO10STEP2
  'Write my name 5 times
```

```
    PRINT "Kuldeep";
  NEXT I
```

DO/LOOP Loop Construct

Syntax:

```
  DO {WHILE|UNTIL} condition
  LOOP
```

or

```
  DO
  LOOP {WHILE|UNTIL} condition
```

Note the two different ways of using it. If you put the condition at the beginning, it is evaluated before each loop execution. If you put it with LOOP at the end, however, it evaluates it *after* each execution! This guarantees that, no matter what, the code inside runs at least once. If you use the WHILE keyword before the condition, the loop runs as long as *condition* is true. If you use UNTIL, the loop runs as long as *condition* is false!

Example:

```
  'This example shows one of its best uses: verifying input!
  DO 'run the code at least once
    INPUT "Enter a number between 1 and 10. ", num%
  LOOP UNTIL num% > 0 AND num% < 11 'wait until it's valid
```

EXIT Command

Syntax: `EXIT {FOR | DO}`

The EXIT command lets you break out of a FOR/NEXT or DO/LOOP construct in the middle of the code. This would be used when you must stop the loop prematurely. For example, if you have a FOR/NEXT going from 1 to 10 and you need to end it early because another condition is true, you can use EXIT FOR to stop the loop.

Example:

```
  DO
    INPUT "Please enter a number between 1 and 10. ",
    num% IF num% > 0 AND num% < 11 THEN EXIT DO PRINT
    "Hey! Can't you read?"
  LOOP    'note that the DO/LOOP can be used without a condition at all,
          'resulting in an infinite loop. EXIT is the only way to break such
a loop.
```