# Programming basics

*By: McTwist*
*May 13, 2018*

## Contents

# 1 Introduction

Learning a programming language is fairly straightforward. This document will go through the basics that will make it easier to learn any language. All languages may have different syntax or special oddities and curls, but in general this document will make sure that the mindset of a programmer will be set and then apply it. Keep in mind that all future documents will contain specialized content toward TorqueScript while this will only be the basics of the basics. No code in here exists in any existing language, but are more used to make generalized algorithms made as a hint of how a program should work.

*Keep in mind that this lesson is only about the theory of programming. Actual TorqueScript code is done in the next lessons.*

# 2 Starting is Slow

Like any language, starting to learn a language is slow and in most cases, troublesome. However, a programming language is a bit easier to learn, but harder to master. The reason for this is that one must first know simple English language, basic maths and advanced logic thinking.

To wrap it around quickly, the English language is good to know as most syntax relies on it. The basic maths is good to know for simple algebraic solutions. One could of course need more advanced maths, but this course does not aim such a thing.

For the advanced logic, one need to know this to be able to tell the computer exactly what it should do. It's extremely rare that a computer is disobedient. If one always assume it will do as told, a program could easily be structured in a logical way to make advanced instructions to the computer.

## 2.1 Variables

First thing to bring up is variables. Think these as labelled containers that will hold any datum one want to store within it. Most commonly this includes, but are not restricted to: *numbers*, *strings* and *objects*.

**Numbers** could either be integer or decimal numbers. These are usually represented with the ordinary base 10 system. One could of course make use of hexadecimal, octagonal and even binary to represent a number. These are left as an exercise to the reader.

**Strings** are used to store text. It could contain anything that can be stored as a character. This includes new line, space, even backspace.

**Objects** will be mentioned more further down.

To store a value to a variable, one could write like so.

```
1: age ← 18
```

This will store the number 18 into the variable `age`. This makes it possible to temporarily store data within the program so the user experience is different depending on the variables content.

This can also be used with strings.

```
1: name ← "Jones"
```

This will store the text `Jones` within the variable `name`. Take a note in the quotation marks, as those are used to define a string. Anything within those quotation marks will be stored within the variable.

## 2.2 Operators

Naturally, one could not make use of the variables unless one got some sort of modifier. In comes the operators. Most of them are maths operators, but there is some that is specific to the computer. This section will only go through the maths ones, which is also called the **arithmetic** operators.

First is the *assignment* (`=`) operator. It was used briefly in previous section to assign a value to a variable. It is visualized with an equal sign. As long as a variable is on the left side, any value on the right will be assigned to it.

```
1: x ← 4
```

Moreover are **addition** (`+`), **subtraction** (`-`), **multiplication** (`*`) and **division** (`/`).

```
1: x ← 4 + 2
2: y ← 4 − 2
3: z ← 4 ∗ 2
4: w ← 4/2
```

This will result in each variable of `x`, `y`, `z` and `w` to contain 6, 2, 8 and 2 respectively. There is also this operator called **modulus** (`%`) that is used to get the remainder from a division. How it is used is left as an exercise to the reader.

Along with the arithmetic operators, there is also the **relational** operators. These are used to compare values. These are **equal** (==), **less than** (<), **greater than** (>), **less equal** (<=), **greater equal** (>=) and **not equal** (!=). Even if they sound somewhat the same, they actually are used in different manners, depending on what one is aiming for.

```
1: 1 = 1
2: 1 < 1
3: 1 > 1
4: 1 ≤ 1
5: 1 ≥ 1
6: 1 ≠ 1
```

Each one of these will give a *boolean* value: `true` or `false`. From the top: `true`, `false`, `false`, `true`, `true`, `false`. To explain this, the first one is 1 equals 1, which they are the same. Then is two false, because 1 is neither bigger or smaller than 1. The two following is true even if it checks for values bigger or smaller, as it also checks if they are the same as well, which they are. The last one is checking if they are not equal, which they are not. Of course one could put in variables in there, to see if its value relation to an another value.

```
1: x ← 1
2: y ← 2
3: y ≠ x
```

This will make the third statement to be true, as the value of `y` (2) is not equal the value of `x` (1).

## 2.3 Conditions

While setting variables and do operations on them, one cannot do anything unless there is some way to change the program flow. In comes conditions.

```
1: x ← 1
2: if x > 0 then
3:     x ← 0
4: end if
```

This will first prepare the variable `x`. Then it will check with if its value is bigger than 0. As this is the case, the `if`-condition will then check if the value is `true`, and if so

it will execute the next statements. The `then` is only used to mark the start of a code block. The `end` is closing the previously mentioned code block. A code block is a bunch of statements grouped together.

It is worth noting that the third statement is **indented**. This is only to visually tell everyone that reads this that this belongs to the statement preceding it. That is, it makes it a lot easier to understand the code-flow. In some languages, indentation is a requirement.

Along with the if-statement, there is two more normal conditions that one should use. They are `else` and `else if`. The `else` is only executed if the preceding conditions were `false`. Of course the `else if` looks like it is `else` and `if` concatenated together, but in most languages they are actually split apart.

```
1: x ← 3
2: y ← 4
3: z ← 3
4: if x = y then
5:     y ← z
6: else if x = z then
7:     z ← y
8: else
9:     x ← y
10: end if
```

We got three variables with values. Two of them are checked if they are equal. If they are, then it will run the next statement. In this case, it is not and will skip the next code block and go for the next condition. Here it is also is false, so it will proceed to the last code block and execute it.

## 2.4 Loops

After getting to know how to change the flow of the program by jumping to statements that should be executed, one could easily make a program like this.

```
1: n ← 1
2: x ← 1
3: x ← x * n
4: n ← n + 1
5: x ← x * n
6: n ← n + 1
7: x ← x * n
8: n ← n + 1
9: x ← x * n
```

However, this is duplicated code, highlighted in red. There is a way to make this way shorter and easier to manage. In comes **loops**.

```
1: n ← 1
2: x ← 1
3: while n ≤ 4 do
4:     x ← x * n
5:     n ← n + 1
6: end while
```

The `while`-statement will repeat its code block until its condition is false. Its condition is checked first, and then each time it repeats its code block. This generates the same result as the previous example.

There is also a loop that make the logic of the example even easier to understand. The `for`-loop is one of the most used loops, as it will easily handle your **iterator**. An iterator is exactly what the variable `n` is doing in the examples. It will increase by a certain amount of number and then be used to run a code block a certain amount of times or get data from a list or variables.

```
1: x ← 1
2: for n ← 1 to 5 do
3:     x ← x * n
4: end for
```

For variable `n`, set the value of 1 and iterate it one step until value is equal or higher than 5. There is several variants of this, but this document will only use this version as it is most used that way.

## 2.5 Functions

Functions are code blocks that one could send in certain variables and execute and then return from where it was called from. The biggest advantages of them is to move out algorithms into them and to reduce duplicate code and reuse them.

```
1: function ADD(x, y)
2:     z ← x + y
3:     return z
4: end function
```

This function can then be called upon.

```
1: x ← 5
2: z ← 4
3: y ← ADD(x, z)
```

Basically, `x` and `z` is set to a value each. Their value is then sent into the function `add`, and now the program flow is moved into there. From there is creates two other variables, and then do some addition and send it into a third variable. Its value is then returned from the function. The program flow will then be moved back to where it was called and the returned value will be assigned to `y`.

Keep in mind here that the variables within add does not affect the variables outside of it. These are called **local variables**. The name of the parameters could be anything, as they are created like normal variables and assigned the values that are sent into the function, in order.

```
1: function SUM(a)
2:     if a < 1 then
3:         return a
4:     end if
5:     return SUM(a − 1)+a
6: end function
```

This function will be called with a number contained within `a`. It will then check if it is less than 1. If not, it will continue and call itself. This will continue until `a` is less than one, which it then will return normally. All accumulates of `a` from each call to sum will then be added together, resulting into `a` sum of 0 up to the value of `a` in the first call to sum.

`sum` is called a **recursive** function. This applies whenever a function calls itself directly or indirectly. Indirectly is done when a function first calls another function that in turn calls the first function.

As previously mentioned, each call to the function will create a new set of local variables that does not affect the previous result. This means that the variable `a` will be created anew with a new value for that function call.

To make this a bit simpler to understand, the previous section example will be used with recursion instead of a `for`-loop.

```
1: function FACTORIAL(n)
2:     if n ≤ 1 then
3:         return n
4:     end if
5:     return FACTORIAL(n − 1)∗n
6: end function
```

This will return the same result as the previous example.

```
1: x ← FACTORIAL(4)
```

Worth noting is that even if recursive functions are really helpful, it is really easy to make it go indefinitely, if not careful. Most languages does not like to call too many functions, and will crash if going too deep.

## 2.6 Objects

Sometimes there is a mess keeping track on variables. It even gets worse when one needs to, for instance, handle several persons with their respective age and name. Here is where **objects** are helpful. Objects are like variables, a container. But instead of values, they contain variables. One could then easily move around groups of data about certain types.

```
1: person ← new Person
2: person.age ← 45
3: person.name ← "James"
```

The first statement will create a `new` object called Person and store it within the variable `person`. Then it will use the **dot**(`.`)-operator to access two of its variables, `age` and `name`, and then assign them values.

Worth noting is that both the `new`-operator and dot-operator is not used in some languages, but are used here to explain how objects are used more easily. For example, some languages uses functions to both create objects and access their variables.

## 3 Assignment

The reader should try to take a look at the factorial math thesis, as that is what this document has gone through, but not explained. It should give a hint on how math can be applied to programming and maybe why it could make some problems hard to solve through programming.

Factorial

As this document is only theoretical, there is no practical assignment.