

Grasshopper VB Scripting Primer

Dr Patrick Janssen

patrick@janssen.name

16th March 2009

Introduction

This booklet was written for students of the Generative Techniques in Design elective at the Department of Architecture, National University of Singapore. Most of the students had no prior experience with programming, so I have attempted to introduce Grasshopper VB Scripting in a way that is as simple and easy to understand as possible. Note that this is a very broad introduction that necessarily glosses over many of the details. This is just a primer.

This is version 1 of this document, and it may continue to evolve. To download the latest version, please go to <http://community.nus.edu.sg/ddm>. In fact, in this current version, there are still a number of sections labelled as [under construction] – these will be updated soon.

The document is based on Rhino3d Version 4 and Grasshopper Version 0.5.0099.

Table of Contents

1	What's it all about	5
1.1	VB.NET.....	5
1.2	Grasshopper Scripting.....	5
1.2.1	Where do I write my script?	6
1.2.2	What about Visual Studio Express?	7
1.2.3	About the code samples	7
2	Writing code.....	8
2.1	Comments	8
2.2	Code.....	8
2.2.1	Statements	9
2.2.2	Variables	9
2.2.3	Control Flow	9
2.2.4	Coding conventions	10
3	Working with Value Types	11
3.1	Some common Value Types.....	11
3.2	Variables and Value Types.....	11
3.2.1	Declaration	11
3.2.2	Assignment.....	12
3.2.3	Combined declaration and assignment	12
3.2.4	Variable use	12
3.3	Some Useful Functions.....	12
3.4	A complete example	13
4	Working with Class Types.....	14
4.1	Classes and Objects.....	14
4.1.1	Classes.....	14
4.1.2	Objects	14
4.1.3	Properties and Methods	15
4.2	Methods in More Detail.....	15

- 4.2.1 Parameters and Arguments..... 15
- 4.2.2 Function Procedures 16
- 4.2.3 Sub Procedures..... 16
- 4.2.4 Constructors 16
- 4.3 Variables and Class Types 16
 - 4.3.1 Declaration 17
 - 4.3.2 Instantiation 17
 - 4.3.3 Assignment..... 17
 - 4.3.4 Combined declaration, instantiation and assignment 18
 - 4.3.5 Parameters and arguments revisited..... 18
 - 4.3.6 Variable Use 18
 - 4.3.7 The dot operator 18
- 4.4 A complete example 19
- 5 Working with Array Types.....20
 - 5.1 Variables of Type Array20
 - 5.2 A complete example20
- 6 Working with String Types20
 - 6.1 Variables of Type String20
 - 6.2 A complete example20
- 7 Control Flow.....20
 - 7.1 For... Next20
 - 7.2 If...Then...Else20
- 8 The VB Script Component21
 - 8.1 The Grasshopper_Custom_Script class21
 - 8.1.1 Imports Statements22
 - 8.1.2 Class declaration23
 - 8.1.3 The first region24
 - 8.1.4 Properties24
 - 8.1.5 Sub Procedures.....24
 - 8.1.6 The second region.....25

8.2	VB Component Inputs and Outputs	26
8.2.1	Inputs parameters.....	26
8.2.2	Outputs parameters.....	27
8.2.3	Debugging code	27

1 What's it all about

[Rhino3d](#) is a popular 3d NURBS based modelling program developed by McNeel.

[Grasshopper](#) is a free plugin for Rhino3d, which allows you to do dataflow modelling. With the Grasshopper approach to dataflow modelling, you can create complex parametric models by defining relationships between entities in your model. These relationships are defined using the 'boxes and arrows' approach. The boxes – called *components* – specify procedures that do something like draw a line. The arrows connect the output from one procedure to the input of another procedure.

Grasshopper provides a two components that allow you to create your own custom procedures using scripting. These are the C# Component and the VB Component. In this document, we will be focusing on the VB Component, for which the user has to write a script in the VB.NET programming language.

1.1 VB.NET

The Microsoft .NET Framework is a software framework for developing programs and applications for the Microsoft Windows operating system. The framework supports a number of different languages, including Visual Basic DotNET and C# DotNET.

Visual Basic DotNET is a full-blown object-oriented programming language. Visual Basic is often referred to using just the initials, VB. The difference between VB and C# is mostly stylistic - the only real difference today is programmer preference. The basis of VB is an earlier programming language called BASIC that was invented by Dartmouth College professors John Kemeny and Thomas Kurtz. VB is easily the most widely used computer programming system in the history of software.

VB is not the same as VB for Applications (VBA) or VB Script. VBA is both a language and an integrated programming environment (IDE), which is embedded into many applications. VB Script is a subset of VBA and is a more rudimentary scripting language to allow users to write simple scripts.

1.2 Grasshopper Scripting

This scripting in Grasshopper should not be confused with the other scripting options available in Rhino3d.

For a long time it has been possible to automate Rhino3d using the RhinoScript scripting language. RhinoScript is written by McNeel and it is based on the Microsoft VBScript language. Since Rhino3, it has also been possible to use VB.NET to write plugins for Rhino. Grasshopper for example is mostly written in VB.NET. However, Rhinoscript and VB.NET plugin development are very

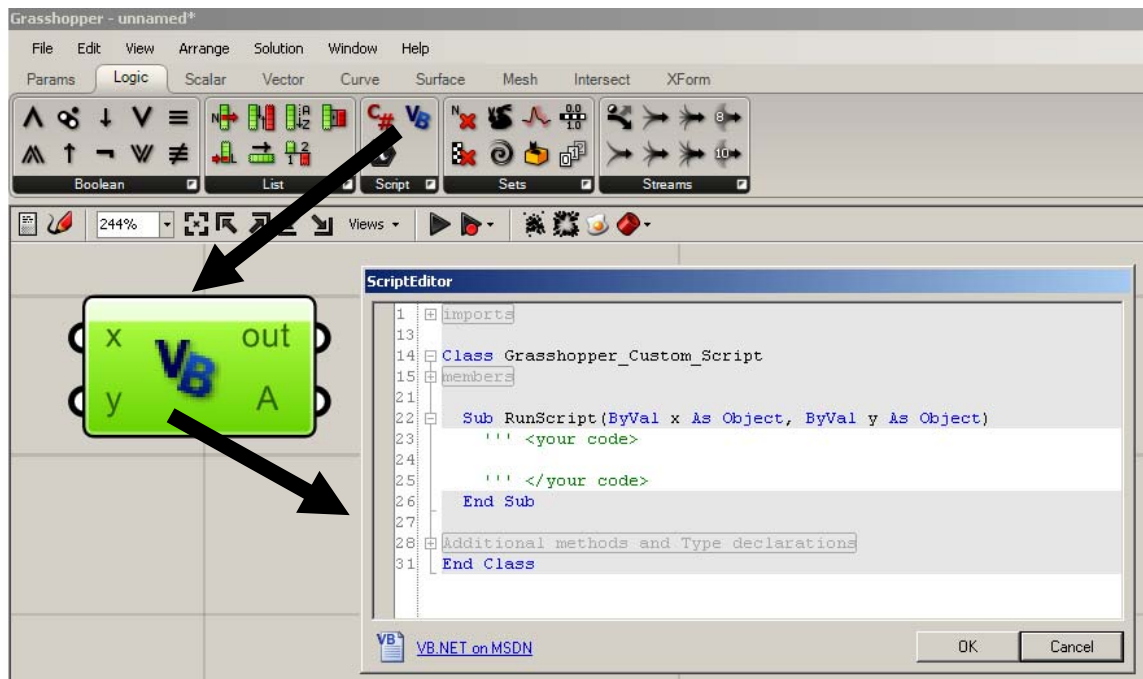
different from writing VB.NET scripts in Grasshopper. (The reason that you cannot use RhinoScript in Grasshopper is because RhinoScript can only operate on objects that are actually inside the document, whereas Grasshopper only draws its geometry into the viewports.)

To summarise:

- 1) Rhino is written in C++ and it exposes a C++ SDK which allows other (i.e. McNeel) people and companies to write plugins for Rhino.
- 2) Using this C++ SDK, someone at McNeel wrote a RhinoScript plugin, which in turn allows people to write scripts for Rhino.
- 3) Using this C++ SDK, someone else at McNeel wrote a DotNET Wrapper plugin, which in turn allows people to write Plugins for Rhino using any DotNET language.
- 4) Using this DotNET wrapper SDK, another person at McNeel (David Rutten) wrote a plugin called Grasshopper.
- 5) Grasshopper allows people to create custom components by writing scripts in either the VB.NET language and the C#.NET language.

1.2.1 Where do I write my script?

For writing a VB script in Grasshopper you use a simple script editor built into Grasshopper. This creates some confusion at first since the documentation on the Internet for VB tends to start with an introduction to the Visual Studio Express IDE.



If you wish to write a DotNET script inside Grasshopper, all you have to do (assuming you already have Grasshopper installed and running) is:

- Open the Logic panel
- Drag a VB component onto the canvas
- Double click the VB component
- Start typing in the white space under where it says ' ' ' <your code>.

The areas in the script window that are greyed out cannot be edited. (Try it – click in the greyed out area and try typing.) When you click OK, Grasshopper will compile your code and (assuming there were no errors during compilation) run it.

1.2.2 What about Visual Studio Express?

If you are writing a VB application, you would use a piece of software for programming called an *Integrated Development Environment* (IDE). For example, for VB there is an IDE called [Visual Studio Express](#) that is free of charge.

However, for writing a VB script in Grasshopper you do not use this IDE. (You should only use the IDE if you intend to write your own components. This is much more difficult.) Instead you use the simple built-in script editor in Grasshopper as described above.

1.2.3 About the code samples

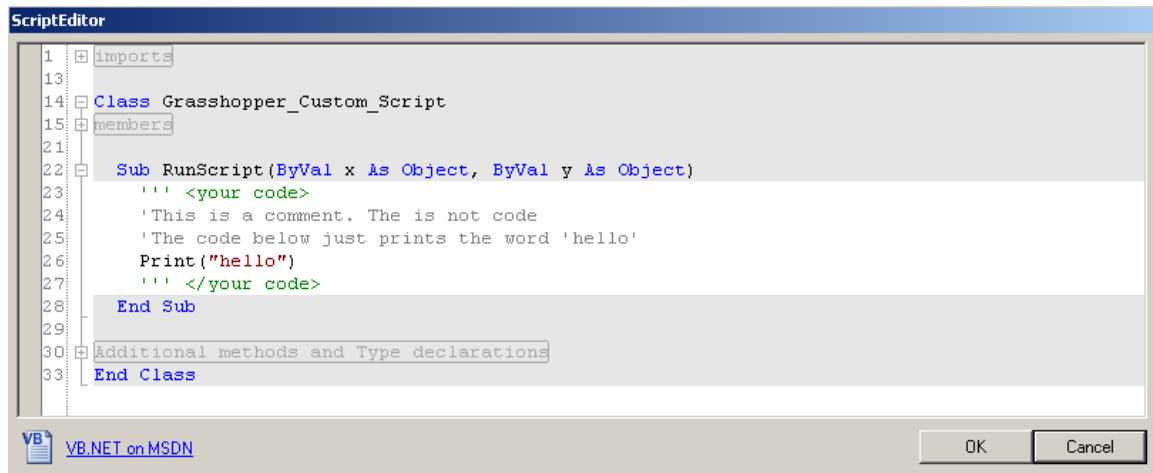
The code samples in the text below are extracts from a longer piece of code. Only those sections with the title 'a complete example' will run if you copy them directly into a script.

With the other bits of code, it takes up too much space to keep repeating the full set of code, so I tend to leave bits out here and there. But the careful reader should easily be able to fill in the missing bits.

(PS At some point, I will improve the code layout to match the format of the Rhinoscript 101 document, which looks much better than this one :)

2 Writing code

Programming in VB involves writing two types of things: comments and code. Here is the same example with some comments and some code.



2.1 Comments

[Comments](#) are lines of text preceded by an apostrophe. These comments are just for you to remind yourself (and maybe others) of what your code is doing. Comments become essential when code gets complex.

The `your code` lines are just comments and can be deleted if you want:

```
''' <your code>
```

```
''' </your code>
```

The reason that there are three apostrophes rather than just one is because there is also a more advanced way of adding comments, called [XML Documentation Comments](#). However for the moment don't worry about this.

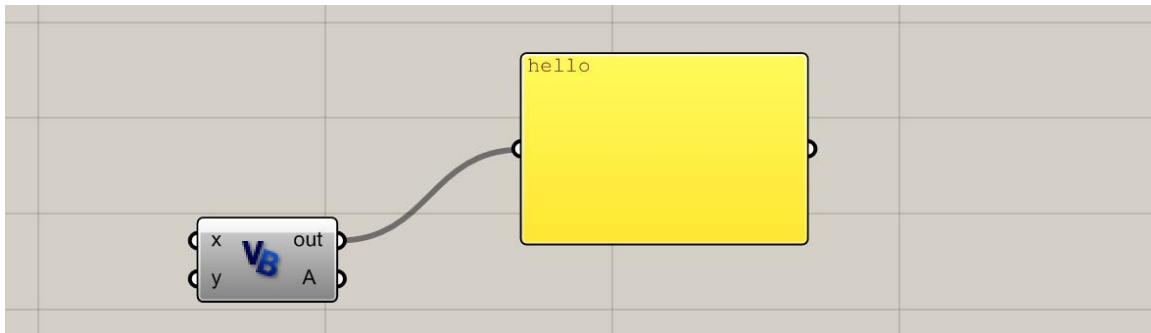
2.2 Code

Code is everything that is not a comment. Code consists of a series of [statements](#) that tell the computer what to do.

In the example above, there is one statement: `Print("Hello")`. This is an execution statement that uses the `Print` function.

The `print` function is a very useful function that displays some text somewhere. (It does not send anything to you printer – when writing computer programs, `print`

means 'display this text'.) In the case of Grasshopper scripts, the text from the print function will always be sent to the *out* parameter of the VB component.



2.2.1 Statements

The three main types of statements that you can write are declaration statements, assignment statements, and execution statements. (Assignment statements are actually a special type of execution statement.)

Declaration and assignment statements are mainly to do with telling the computer to create and set variables, where a variable is a name that can be associated with some value.

Execution statements are mainly to do with telling the computer to actually do something. An important type of execution statement are things like functions, loops, and if statements, which are referred to as [control flow](#) statements.

2.2.2 Variables

Variables may be associated with simple types of things like numbers, and more complex types of things like points and lines. These are referred to as [Value Types and Reference Types](#) respectively. For Reference Types, there are a number of different types – three Reference Types will be introduced:

- Section 4 will introduce Class Types,
- Section 5 will introduce Array Types, and
- Section 6 will introduce String Types.

2.2.3 Control Flow

In the absence of any control flow statements, a computer will step through the statements that it finds in the program file in a sequential manner, from beginning to end, executing each statement in turn.

Control flow statements allow you to regulate the flow of your program's execution. For example, you might want to repeat an action ten times. Rather

than repeating the code ten times you can insert a loop structure into your code that will tell the computer to repeat a particular piece of code a certain number of times.

- Section 7 will introduce Control Structures

In general, the two main types of control structures are [decision structures](#) and [loop structures](#). One of each will be introduced: for decisions, If...Then...Else construction will be introduced; for loops the For... Next construction will be introduced.

2.2.4 Coding conventions

Code conventions focus on the stylistic aspects of [naming](#) and [coding](#). For example, should you start a variable with an upper or lower case letter? (the answer is lowercase). For those new to programming, this type of thing may seem unimportant. But as you do more programming, you realise that it is very important for legibility of code. If you come back to some code a few months later, it is much easier to understand if you have followed some kind of consistent coding conventions. As far as possible, this document follows the standard code conventions.

One particular point to note is the use of the [line-continuation character](#), which is an underscore (`_`). This is used when a long line of code needs to be broken down into smaller lines. Due to the formatting of this document, this has been used in a number of places to fit the code onto the page.

3 Working with Value Types

A variable is identified by its name. The name of a variable cannot be the same as one of the language [keywords](#), which are reserved for other purposes. But apart from that, the variable name can be almost anything, like `x`, or `myVar`, or `blabla`. By convention, variables use what is referred to as `mixedCase`: they start with a lowercase letter and each subsequent word starts with an uppercase letter.

3.1 Some common Value Types

Assigning a data type to a variable makes it easier for the computer to figure out how the variable can—or can't—be used. Think of it this way: if you had three variables, two of which held numbers while the third held a name, you could perform arithmetic using the first two, but you can't perform arithmetic on the name.

Some of the most important value [data types](#) are the following:

Boolean	True	A value that is either true or false
Integer	20	Whole numbers less than 2 billion
Long	2e9	Whole numbers larger than two billion
Double	20.2	Numbers with a fractional part
Char	A	A single character

So when you see the word `Double` in a script, then this is not some function to double some number, it is actually just a type.

3.2 Variables and Value Types

A variable can store simple types of things like numbers. For example, I can say, I have a variable whose name is `x` and who can store a number of type `Integer`. I can then say that my variable `x` should store the value `20`.

These two steps are called declaration and assignment. First, you have to declare the type of each variable, and second you actually assign a value. (You will see later that you can combine declaration and assignment into a single line.)

3.2.1 Declaration

A declaration is used to declare the type of variable, using a [Dim statement](#). For example:

```
Dim p As Integer
```

This statement declares that the variable named `x` can store an `Integer` value.

3.2.2 Assignment

Once you have declared the type of variable, you can then assign a value to it. For example:

```
p = 20
```

3.2.3 Combined declaration and assignment

In order to save space, you can put declaration and assignment on a single line. However, it is still important to conceptually think of these as two separate steps. Combined declaration and assignment looks like this:

```
Dim p As Integer = 20
```

3.2.4 Variable use

From now on, the variable with name `x` stores the value 20. For example:

```
Print(p)
```

This `Print` function would print 20. Another more complex example is this one:

```
p = p + 10  
Print(p)
```

This time, the value of the variable `x` would be increase by 10. The `Print` function would print 30. This example uses two operators, and addition operator and an assignment operator. There are [many operators](#) that can be used with Value Types.

3.3 Some Useful Functions

The [Print function](#) is one of the built in VB run-time member functions. There are [many other useful VB functions](#).

A function has one or more inputs and a single output. The function will process the inputs and produce the output. The inputs are referred to as *arguments* and the output is referred to as the *return value*. The format for calling a function is something like this:

```
returnValue = FunctionName(argument1, argument2)
```

For example, there are a useful set of functions to convert one simple data type into another, known as [type conversion functions](#). One of the functions is called `CStr` and converts things like `Integer` or `Double` to `String`.

```
Dim someText As String  
someText = CStr(p)
```

3.4 A complete example

Here is a complete example that creates some `Double` values and converts one of them to a `String` to be printed:

```
Dim p As Double  
Dim q As Double  
Dim r As Double  
p = 20  
q = p + 10  
r = p / q  
Print("The value of r is " + CStr(r))
```

This script will print:

The value of r is 0.6666666666666667

4 Working with Class Types

As well as these simple data types, variable can also be associated with more complex types of entities called [objects](#). For example, one common type of object in Grasshopper is an object of type `On3dPoint`, which is used for representing points in space in Rhino. Another is `OnLine`, which is used for representing Lines in Rhino.

4.1 Classes and Objects

A class is a representation of a type of object. You can think of it as the object's [blueprint](#). Just as a single blueprint can be used to build multiple buildings, a single class can be used to create multiple copies of an object. The names `On3dPoint` and `OnLine` are actually names of classes.

When talking about classes and objects, you generally say the object A is of *type* B (where B is a class), or alternatively A is an *instance* of class B.

4.1.1 Classes

When you are writing VB DotNET scripts in Grasshopper, you will not be creating any of your own classes. (Advanced users may create their own libraries of classes, but presumably they will not need to read this primer.) However, you need to have a good basic understanding of classes and objects so that you can use the existing libraries in the DotNET framework and in Rhino. In fact, one of the hardest things in programming is learning how to use these existing class libraries since there are thousands of these classes.

Class libraries consist of large numbers of compiled classes that are packaged up into files such as .exe files or .dll files. These packages are referred to as [Assemblies](#). Inside these assemblies, there may be hundreds of classes, so there has to be a way of organising all these classes. Not surprisingly, they are organised into a hierarchy of groups and sub-groups that you can think of like the folders where you store files on your computer. These folders are referred to as [Namespaces](#). For example, `On3dPoint` and `OnLine` are both in the Assembly `xxx.dll` and in the Namespace `RMA.OpenNURBS` (where `RMA` is one Namespace, and `OpenNURBS` is a sub Namespace).

Note that the reason that the classes in these examples start with the letters 'On' is because they are part of the `OpenNURBS` library. Resist the temptation of thinking that the class `OnLine` has something to do with being on a line. Think of it as 'Open NURBS line'.

4.1.2 Objects

Objects are always instances of some type of class. Objects have

- properties that describe their attributes, and

- methods that define their actions.

The properties and methods that an object has are defined by its class.

4.1.3 Properties and Methods

The properties are like variable that are inside the object. For example, an object of type `On3dPoint` has `x`, `y` and `z` properties that hold data for the co-ordinates of the point. In this case these variables are simple types of type `Double`.

Another example is an object of type `OnLine`. In this case it has two properties, `from` and `to`, which are the start point and the end point of the line. However, in this case these variables are themselves class types of type `On3dPoint`.

The methods are like [procedures](#) that are inside the object. Such procedures are similar to the VB run-time member functions discussed earlier. For example, objects of type `On3dPoint` have a Function Procedure method called `DistanceTo` that calculates the distance to another point. Another example is objects of type `Online` have a Function Procedure method called `Length` that calculates the length of the line.

Object methods are described below in more detail.

4.2 Methods in More Detail

There are two types of procedures: [Function Procedures](#) and [Sub Procedures](#). Both of these types of procedures can accept input data, which is specified by the parameters for the procedure. Only Function Procedures can produce output data, which is specified as the return value of the Function Procedure.

4.2.1 Parameters and Arguments

Often procedures need some data as an input. This input is described in terms of [parameters and arguments](#).

A *parameter* represents a value that the procedure expects you to supply when you call it. The procedure's declaration defines its parameters. An *argument* represents the value you supply to a procedure parameter when you call the procedure. The calling code supplies the arguments when it calls the procedure.

The one thing that is a little complex to understand for beginners when dealing with parameters and arguments is the [passing mechanism](#). This actually specifies how the arguments are passed to the procedure. There are two options: they can be passed either *by reference* (specified by the `ByRef` keyword) or *by value* (specified by the `ByVal` keyword). For the moment, we will ignore what this means exactly and return to it later.

4.2.2 Function Procedures

Function Procedure are the same as the previously described functions: they have one or more *parameters* and a single *return value*. The implementation of a Function Procedure to add two numbers of type `Double` might look something like this:

```
Function add2(ByVal num1 As Double, ByVal num2 As Double) _
As Double
    Return num1 + num2
End Function
```

This function should be fairly self explanatory. The inputs are two parameters called `num1` and `num2`, both of type `Double`. The output is of type `Double` (and for the output, there is no need to specify a name). Inside the function, the `Return` statement returns the result of adding the two numbers together.

4.2.3 Sub Procedures

Sub Procedures are almost the same as Function Procedures, except that they don't return anything. The implementation of a Sub Procedure to print "Hello " with some name might look something like this:

```
Sub sayHello(ByVal name As String)
    Print("Hello " & name)
End Sub
```

4.2.4 Constructors

A class must also specify one or more special Sub Procedure methods called [Constructors](#) that are used to create an object instances. These Sub Procedures are different from other Sub Procedures in that the Sub Procedure name is always the word `New`, and they are only executed when the object is first created.

4.3 Variables and Class Types

Declaration and assignment of variables for class types would seem to be very similar to value types. However, there are actually some very fundamental differences, particularly when it comes to assignment.

When working with value types a variable will store the actual value. So if `a` and `b` are `Doubles` and you write `b = a`, then the computer will take the value stored in `a` and copy it into `b`.

However, when working with complex types, a variable only stores a reference to an object, but does not store the object itself. (You may have noticed that with objects, I wrote that the variable 'points to' the object.) You can think of the reference as an address – the variable stores the address of the object. So if `pt1`

and `pt2` are `On3dPoint` objects and you write `pt2 = pt1`, the computer will take the object reference stored in `pt1` and copy it into `pt2`.

4.3.1 Declaration

As with value types, the same [Dim statement](#) can be used to declare class types. For example:

```
Dim pt1 As On3dPoint
Dim pt2 As On3dPoint
```

This statement declares that the variable named `pt1` can point to an object of type `On3dPoint`. This is actually the same as for value types.

4.3.2 Instantiation

For value types, there is not a separate instantiation step since value types hold the data directly inside the variable name. However, for reference types, the data is created somewhere else, and the variable only holds a reference to that data. This means that the data first has to be created, which is called *instantiation*, and is done using the class's constructor method.

To instantiate a new object, you have to use the [New keyword](#). When the computer sees this keyword, it will then know to look for one of the constructor methods to instantiate a new object from the class. For example, lets say you want to create a new point with coordinates (20,30,40):

```
pt1 = New On3dPoint(20,30,40)
```

In this case, the `New` keyword is used to indicate that you want to create a new object using one of the constructor methods of the class `On3dPoint`.

4.3.3 Assignment

Assignment looks exactly the same as for value types but remember that in the background something very different is going on.

For example, you could do the following:

```
pt2 = pt1
```

There is a subtle trap that new programmers will tend to fall into when dealing with complex variables. Since the actual object is not copied, if you change a property of `pt1`, then the property of `pt2` will also change.

```
pt1 = New On3dPoint(20,30,40)
pt2 = pt1
pt1.x = 22.5
Print(pt2.x)
```

The `Print` function will print 22.5. This is not the case with simple variable.

```
p = 20
q = p
p = 22.5
Print(q)
```

The `Print` function will print 20.

4.3.4 Combined declaration, instantiation and assignment

As with Value Types, with Reference Types you can put both declaration and assignment on a single line. For example:

```
Dim pt2 As On3dPoint = pt1
```

For Reference Types, you can also create a single line that includes three separate steps: declaration, instantiation and assignment.

```
Dim pt2 As New On3dPoint(20,30,40)
```

4.3.5 Parameters and arguments revisited

[under construction – discuss `ByRef` and `byVal`]

4.3.6 Variable Use

A variable whose value is an object can be assigned in a similar way as a simple variable. (For example see `pt2 = pt1` above.)

However, for other [operators](#) such as addition, this may not be so straight forward. In fact, this depends on the way the class has been implemented. Some classes provide special functionality so that instances of these classes can be used together with the standard operators.

For example, the `On3dPoint` class can be used with [arithmetic operators](#) such as addition, subtraction and division. For example, the mid-point half way between two other points can be found as follows:

```
Dim midPt As On3dPoint
midPt = p1 + (pt2 - pt1 / 2.0)
```

4.3.7 The dot operator

With objects, it is possible to access the properties and methods inside these objects using a full stop between the name of the variable and the name of the property or method. This full stop is called the [dot operator](#).

For example, the `x` property of a `On3dPoint` can be accessed like this:

```
Dim val As Double
val = pt1.x
```

This code first declares a new variable of type `Double` and then sets the value of this variable to the same as the `x` property of the point `pt1` (which in this case is 20).

The dot operator is used in the same way to access the methods inside the object. So for example, to measure the distance between two points you can use the `DistanceTo` method which exists inside any object of type `On3dPoint`:

```
Dim dist As Double
dist = pt1.DistanceTo(pt2)
```

The dot operator can also be used to chain together a series of objects. For example, consider a line object. The line object will contain (as properties) two point objects, and each point object will contain (as properties) `x`, `y`, and `z` `Double` values. In the example below the `x` co-ordinate of the start of the line is accessed using `ln.from.x`:

```
pt1 = New On3dPoint(0,0,0)
pt2 = New On3dPoint(20,30,40)
Dim ln As OnLine
ln = New OnLine(pt1, pt2)
ln.from.x = 50
```

4.4 A complete example

Here is a complete example that creates a line between two points:

```
`declare two points called pt1 and pt2
Dim pt1 As On3dPoint
Dim pt2 As On3dPoint

`assign pt1 by creating a new point
pt1 = New On3dPoint(20, 30, 40)

`assign pt2 by creating a copy of pt1
pt2 = New On3dPoint(pt1)

`move pt2 by changing one of the co-ordinates
pt2.x = 0

`declare a line called ln
Dim ln As OnLine

`assign ln by creating a new line
ln = New OnLine(pt1, pt2)
```

5 Working with Array Types

[under construction]

5.1 Variables of Type Array

[under construction]

5.2 A complete example

[under construction]

6 Working with String Types

[under construction]

6.1 Variables of Type String

[under construction]

6.2 A complete example

[under construction]

7 Control Flow

[under construction]

7.1 For... Next

[under construction]

7.2 If...Then...Else

[under construction]

8 The VB Script Component

As described previously, the VB Component can be found in the *Logic* panel. The Help for this component states the following:

This component attempts to compile and run user specified VB.NET code. By default, the component contains no code. You can supply code by double-clicking the component or by picking the "Edit..." menu item.

Any errors that occur during compilation are sent to the 'out' output parameter and the component menu.

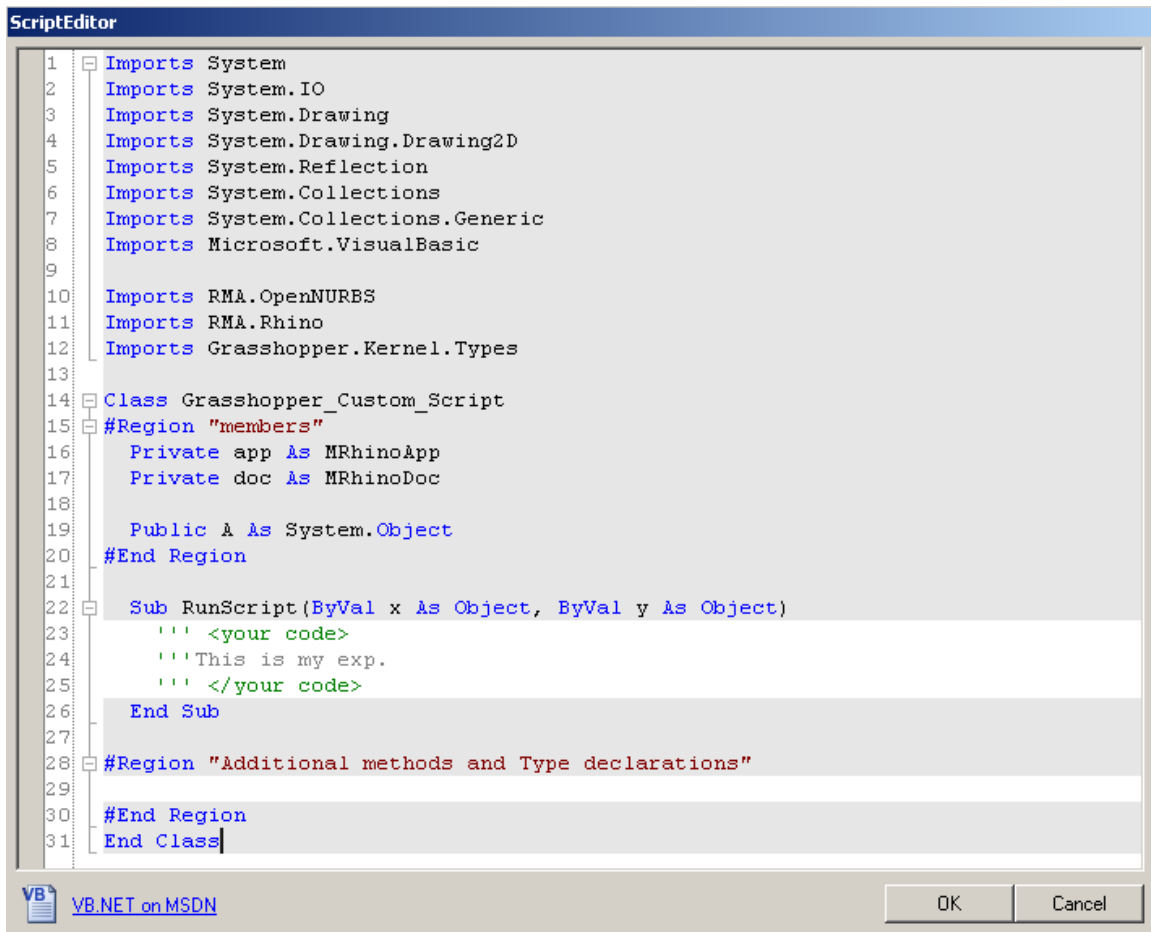
By default, there are two input parameters {x,y} and one output parameter {A}, all of which are of type System.Object. Input and output parameters can be added, deleted and renamed through the component menu.

In addition to this, there are also some other remarks at the bottom of the help text, some of which will be discussed below.

8.1 The *Grasshopper_Custom_Script* class

When a user enters code into the VB component, the code is placed inside a class template called `Grasshopper_Custom_Script`. When the user clicks OK on the script editor, the code is then compiled by Grasshopper and stored in memory. In this section this class template will be described. To see the full template, expand the hidden sections by clicking the + links.

The `Grasshopper_Custom_Script` is a typical VB class and consists of some Imports statements, some Properties, and one Sub Procedure. You also have the option of adding additional Sub Procedures or Function Procedures to the class.



8.1.1 Imports Statements

At the top of the template, you will see a set of lines all starting with the word `Imports`.

As mentioned previously, class libraries are packaged into Assemblies, and the classes inside these Assemblies are organised into hierarchical Namespaces. In order to use a particular class in some Assembly, you first need to set up your development environment so that the compiler knows about that Assembly. You can then use any class in the Assembly as long as you state the Namespaces in front of the class name. This is referred to as *qualification*. For example, to declare a new point, you can write:

```
Dim pt1 As RMA.OpenNURBS.On3dPoint
```

In theory this works fine. In practice, it is a real pain to have to write so much text. The [Imports statement](#) tells the computer where to look for classes that have no Namespace. This means that the classes in the imported Namespaces can be named in the code without qualification. So, for example:

```
Imports RMA.OpenNURBS  
Dim pt1 As On3dPoint
```

Currently, there is no way of adding additional Assemblies and Imports statements to your script in the VB scripting component. However, by default, many of the most useful Namespaces from both the DotNET libraries and from the Rhino libraries are already imported.

The DotNET Namespaces that are imported are:

- The `System` Namespace plus and six others under this Namespaces.
- The `Microsoft.VisualBasic` Namespace.

The MSDN website has a [Class Library Reference](#) that lists all the classes that are available in these Namespaces. (Grasshopper is compiled using VB 2008 Professional and the DotNET 2.0). For each class, the class members are listed (such as class properties, class methods, and class constructors). The documentation is quite good.

The Rhino Namespaces that are imported are:

- The `RMA.OpenNURBS` and the `RMA.Rhino` Namespaces.
- The `Grasshopper.Kernel.Types` namespace.

For the Rhino libraries, there is no equivalent website to the MSDN website. However, you can download the Rhino 4.0 .NET Framework SDK, and inside the zip file, you will find a file called `RhinoDotNetDocs.chm` that is a windows help file that contains the documentation for RMA classes. In fact, the documentation is very limited – it consists mostly of just the names of the classes and class members.

As a beginner, the best way to approach these class libraries is through examples. On the Grasshopper website, there is a page with [scripted examples](#) where you can see how various classes are used. You can then look up these classes in the documentation, and try and figure out why the classes are being used the way they are. The opposite approach, where you start with the class libraries, is very difficult due to the complexity of the classes and is only really feasible if you are an experienced programmer.

8.1.2 Class declaration

After the Imports statements, the next line in the template is the `Class` declaration. This declaration simply tells the computer that there is a class called `Grasshopper_Custom_Script`.

8.1.3 The first region

After the class declaration, the next section of code is enclosed between two lines: `#Region "members"` and `#End Region`. This is a [Region Directive](#) which actually has nothing to do with the script. It is purely a visual thing, in that it allows bits of the code to be hidden and shown using the + and – links.

8.1.4 Properties

Inside the region called `members`, a set of class properties are declared. By default, there are three properties:

```
Private app As MRhinoApp
Private doc As MRhinoDoc
Public A As System.Object
```

As discussed previously, properties are simply variables that exist inside a particular object. So, for example `On3dPoint` has the properties `x`, `y` and `z`. In this case, the template has defined three properties:

- `app` gives you access to the Rhino application. For example, your script might want to check which viewports are visible.
- `doc` gives you access to the Rhino file that is currently open. For example, your script might want to read geometry directly from this file.
- `A` is linked to the output of the VB Script component. Inside your script, whatever you set `A` to will be the output of the `A` parameter.

The last one is the most important one since this is what lets you send some data out of your component. There is one issue that needs to be addressed regarding the type of this property. The template sets the type of `System.Object`, and later we will see that this type cannot be changed. At first this may seem odd, because what if you want to output a different type of object such as a `On3dPoint`, or even just a simple `Integer`. The reason this is not a problem is due to something called class [inheritance](#). Up to now, the issue of class inheritance has been side-stepped, mainly because it is a little complex.

For now, the only thing that really needs to be understood is that classes are actually used to define a whole hierarchy of categories and sub-categories. Classes in the same category share certain types of information. Right at the top of the hierarchy is the class `System.Object`. This means that all class must always be of type `System.Object`, which in turn means that the property called `A` in the template can be set to anything.

8.1.5 Sub Procedures

After the properties declared within the `members` region, the next lines of code declare the main [Sub Procedure](#) inside which you can write your script, called

RunScript. By default, the Sub Procedure declaration in the template is as follows:

```
Sub RunScript(ByVal x As Object, ByVal y As Object)
    ''' <your code>

    ''' </your code>
End Sub
```

The main thing to note about this Sub Procedure is the relationship between the parameters in the Sub Procedure declaration, and the inputs to the VB Script component in Grasshopper. Although this Sub Procedure declaration is not editable here in the script editor, we will see below how the parameters can be edited using the Grasshopper interface.

8.1.6 The second region

After the end of the RunScript Sub Procedure, there is a second region with the description Additional methods and Type declarations. Apart from the area inside the RunScript Sub Procedure, this is the only other area of the template that is editable.

This second region is where you can write your own additional Sub Procedures and Function Procedures. In most cases, it only really makes sense to create Function Procedures, and not Sub Procedures. More advanced users may also create type your own [user-defined data types](#) here.

Function procedures can be used to capture some code that gets repeated numerous times in your script. For example, you might be writing a script that requires various random points of type On3dPoint. In this case, you could write two functions: one that returns a random number (where the value is between some lowerbound and upperbound) and another that returns a random point (where the x, y and z co-ordinates are between some lowerbound and upperbound).

```
`Function that returns a random number of type Double
Function rndNum(ByVal low As Double, ByVal upp As Double) _
As Double
    Return ((upp - low) * Rnd()) + low
End Function
```

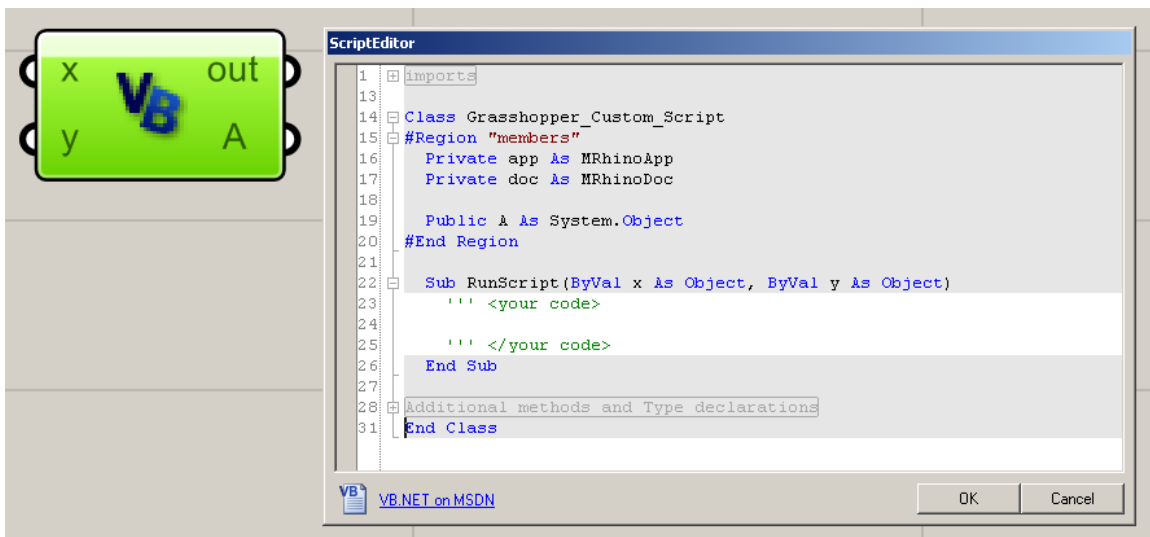
```
`Function that returns a random point of type On3dPoint
Function rndPt(ByVal low As Double, ByVal upp As Double) _
As On3dPoint
    Dim x As Integer = rndNum(low, upp)
    Dim y As Integer = rndNum(low, upp)
    Dim z As Integer = rndNum(low, upp)
    Dim pt As New On3dpoint(x, y, z)
    return pt
End Function
```

8.2 VB Component Inputs and Outputs

As will all components in Grasshopper, VB Components have input parameters and output parameters. These can be used for connecting the scripted component to other components. These other components may be either built-in Grasshopper components or scripted components.

8.2.1 Inputs parameters

The input parameters specify the inputs into the component. By default, there are two parameters, *x* and *y*. Looking at the image below, you will notice that there is a relationship between these two inputs and the two arguments to the `RunScript` Sub Procedure.



Input parameters can be added, deleted and renamed through the VB Component menu. Right click the component, and select *Input parameters* from the drop down menu. Any changes that you make here will be reflected in the `RunScript` method declaration. For example, if you rename the two input parameters from *x* and *y* to *pt1* and *pt2*, then the methods declaration will be automatically updated to the following:

```
Sub RunScript(ByVal pt1 As Object, ByVal pt2 As Object)
    ''' <your code>

    ''' </your code>
End Sub
```

If you are certain that a specific input parameter always provides data of the same type (say, `On3dPoint`), then you can specify a type-hint for that parameter. Type-hints improve error checking and performance. To specify a type hint, right click the component, select the input you want to change from the drop down

menu, and then select *Type hint* from the drop down list, and specify the type. Any changes that you make will be reflected in the `RunScript` method declaration. For example, if you change the types for `pt1` and `pt2` to `On3dPoint`, then the methods declaration will be automatically updated to the following:

```
Sub RunScript(ByVal pt1 As On3dPoint, ByVal pt2 As On3dPoint)
    ''' <your code>

    ''' </your code>
End Sub
```

8.2.2 Outputs parameters

Output parameters can be added, deleted and renamed in the same way as input parameters, through the VB Component menu. Any changes that you make will be reflected in the declaration of the class properties. For example, if you rename the output parameter from `A` to `line`, then property declarations will be updated to the following:

```
#Region "members"
    Private app As MRhinoApp
    Private doc As MRhinoDoc

    Public line As System.Object
#End Region
```

Output parameters do not support type-hints at this point.

8.3 Creating your own components

[under construction]

8.3.1 Design issues

[under construction]

8.3.2 Debugging

[under construction]

8.3.3 Exporting

[under construction]

Acknowledgments

Parts of this document have been extracted from posts by David Rutten (McNeel) to the Grasshopper user group.