Red Centre Software
Data Comes Alive

Beginner Guide to VB for Ruby

Version: 23 May 2016

Confidential.

Beginner Guide to VB for Ruby

This document explains the essential concepts required for Ruby scripting using Microsoft's VB (Visual Basic) platforms. VBScript is used initially, because it has the simplest syntax and runs directly from Ruby. Some examples are then modified for VB for Applications (using Excel) and VB.Net.

The approach taken is practical rather than theoretical, since only elementary programming concepts are required. The aim is to be able to understand, edit and write Ruby scripts which do something useful as soon as possible.

VB SCRIPT

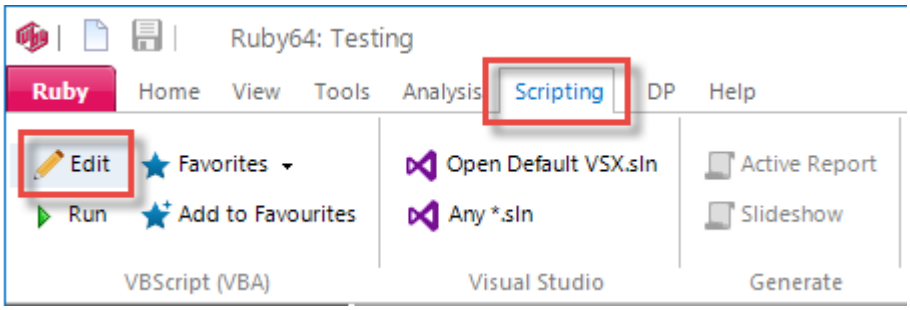
Hello World

The traditional starting point is to get your computer to put up a message saying "Hello World". The purpose of this is to ensure that all the system components are in place for your computer to respond to scripted instructions and do something.

The built-in scripting platform uses VB Script. The VB Script engine is actually part of Windows itself, so nothing extra needs to be installed.

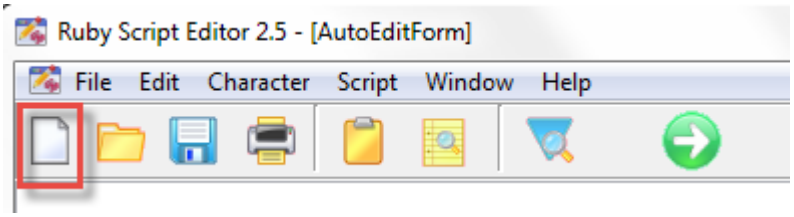
To start the editor

- Scripting | Edit



To start a new scripting document

- File | New or click the New icon on the local toolbar

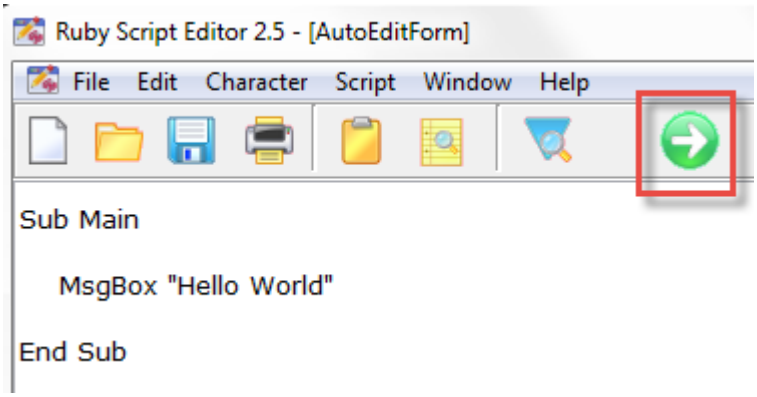


A blank document opens.

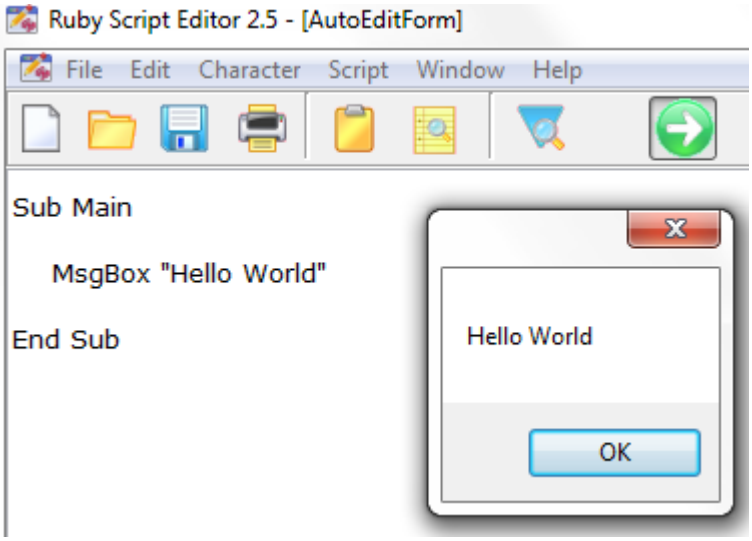
- Type exactly this (or copy/paste the three lines below):

```
Sub Main
    MsgBox "Hello World"
End Sub
```

- Click the Run button



You should see



This simple exercise tells you that:

- The starting point for a VB program is *Sub Main*
- The *MsgBox* command takes a string parameter
- String literals are double-quoted
- The end point is marked by *End Sub*

Note that double quotes must ALWAYS be "…" and NEVER "…". This is most often a problem when copy/pasting VB code which has come through a text processor (eg Word) which defaults to sloping quotes. All code examples in this document should be copyable. By *code*, we mean lines of VB script, and not a survey category code like 1=Under 18 (and *coding* means to write a script, not assign categorical codes to verbatims).

Variables and Assignment

A variable in the VB context means a named value (most commonly numeric or string) where the name is fixed, but the value is not. Variable names must start with an alpha, and have no punctuation other than underscore. You can otherwise call a variable anything you want as long as your name does not conflict with a name already exposed in the VB platform – for example, you cannot use *main* or *sub* or *msgbox* or *end*. Variable names are case-independent.

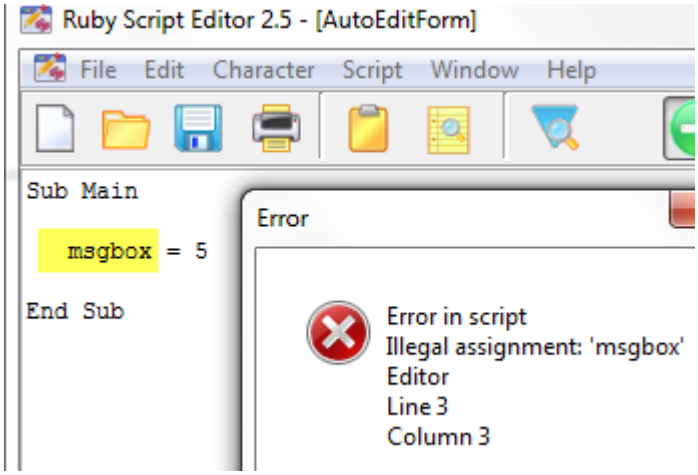
This example assigns the value 5 to a variable called *myvar* and then displays it:

```
Sub Main
   myvar = 5
   MsgBox myvar
End Sub
```

This example assigns the value "I like cats" to a string variable called *mystrvar* and then displays it:

```
Sub Main
   mystrvar = "I like cats"
   MsgBox mystrvar
End Sub
```

This example gives an error, because *msgbox* is a reserved word (or key word):



Since variable names (and routine names) are case-independent, you could write

mystrvar = "I like cats"

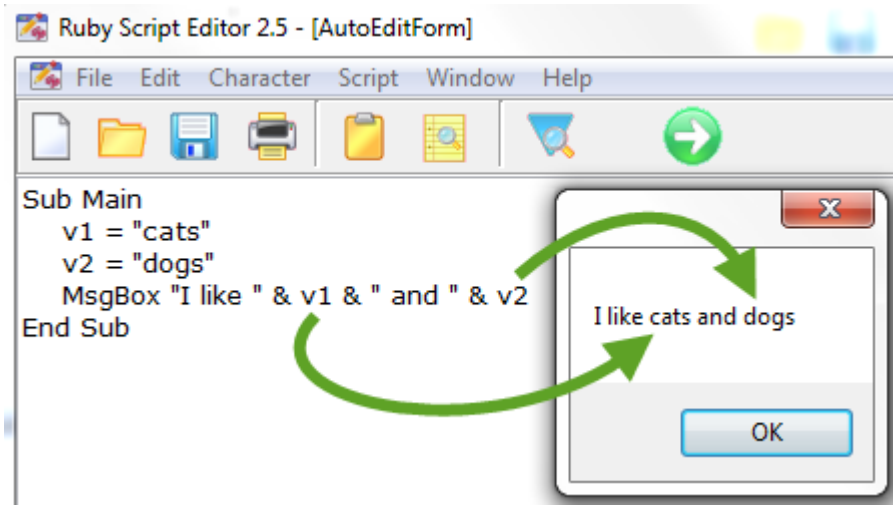
msgbox MyStrVar

This is needlessly confusing, of course, so it is always a good idea to self-enforce consistent casing as a work practice.

String Concatenation

The string concatenation operator is &.

```
Sub Main
    v1 = "cats"
    v2 = "dogs"
    MsgBox "I like " & v1 & " and " & v2
End Sub
```



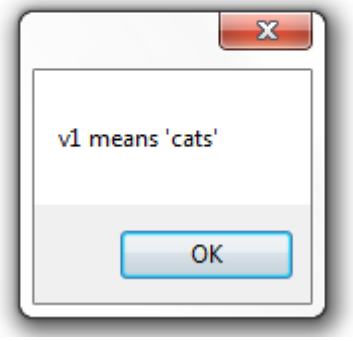
"cats", "dogs", "I like " and " and " are called string literals. v1 and v2 are called string variables.

Conditionals

If...Then

This is the simplest condition. The entire If…Then must be on a single line.

```
Sub Main

   v1 = "cats"
   If v1 = "cats" then MsgBox "v1 means 'cats'"

End Sub
```



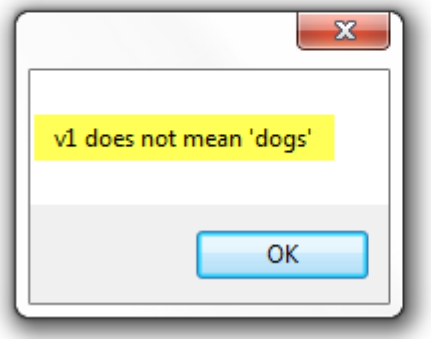
Note that the equality operator in VB is the same as the assignment operator.

```
myvar = 5
If myvar = 5 then…
```

The = sign has two meanings. In the first usage, the value of 5 is assigned to or stored in the variable *myvar*. In the second usage, *myvar* is tested against 5 for equality. The VB engine knows which is which from the context.

If…Then…Else

```
Sub Main

   v1 = "cats"
   If v1 = "dogs" Then MsgBox "v1 means 'dogs'" Else MsgBox "v1 does not mean 'dogs'"

End Sub
```



If the *Then* clause is not true, the *Else* clause is executed instead. The entire *If…Then…Else* must be on a single line.

If…Then…End If

If more than one line is needed, then you use *End If* to mark the end of the block.

```
Sub Main

    v1 = "cats"
    v2 = "dogs"
    flag = true

    If flag Then
        MsgBox "v1 means " & v1
        MsgBox "v2 means " & v2
    End If
```

```
End Sub
```

If…Then…Else…End If

```
Sub Main

    v1 = "cats"
    v2 = "dogs"
    flag = true

    If flag Then
        MsgBox "v1 means " & v1
        MsgBox "v2 means " & v2
    Else
        MsgBox "the value of flag is false"
    End If

End Sub
```

If…Then…ElseIf…Else…End If

```
Sub Main

    flag1 = false
    flag2 = true

    If flag1 Then
        MsgBox "flag1 is true"
    ElseIf flag2
        MsgBox "flag2 is true"
    Else
        MsgBox "flag1 and flag2 are both false"
    End If

End Sub
```

Note that *ElseIf* does not have a space, whereas *End If* does.

Boolean Operators

The common logical operators are the full words *And*, *Or* and *Not*.

```
Sub Main

    flag1 = false
    flag2 = true

    If flag1 And flag2 Then
        MsgBox "both flags are true"
    ElseIf flag1 Or flag2
        MsgBox "one or the other is true"
    ElseIf Not (flag1 And flag2)
        MsgBox "both flags are false"
    End If

End Sub
```

Note the parentheses in *ElseIf Not (flag1 And flag2)*. *(flag1 And flag2)* is first evaluated to either true or false, which is then reversed by *Not* to either false or true.

Relational Operators

The relational operators are

< less than

> greater than

<= or =< less than or equal to

>= or => greater than or equal to

= equal to

```
Sub Main

    val1 = 5
    val2 = 10

    If val1 < val2 Then
        MsgBox val1 & " is less than " & val2
    ElseIf val1 = val2 Then
        MsgBox val1 & " is the same as " & val2
    ElseIf val1 > val2 Then
        MsgBox val1 & " is greater than " & val2
    ElseIf val1 <= val2 Then
        MsgBox val1 & " is less than or equal to " & val2
    ElseIf val1 >= val2 Then
        MsgBox val1 & " is greater than or equal to " & val2
    Else
        MsgBox "Could not evaluate the relation"
    End If

End Sub
```

Arithmetic Operators

All standard operators are supported:

+ addition

- subtraction

* multiply

/ divide

^ raise to the power of

Mod remainder after division, 6 Mod 5 = 1

() order of operations

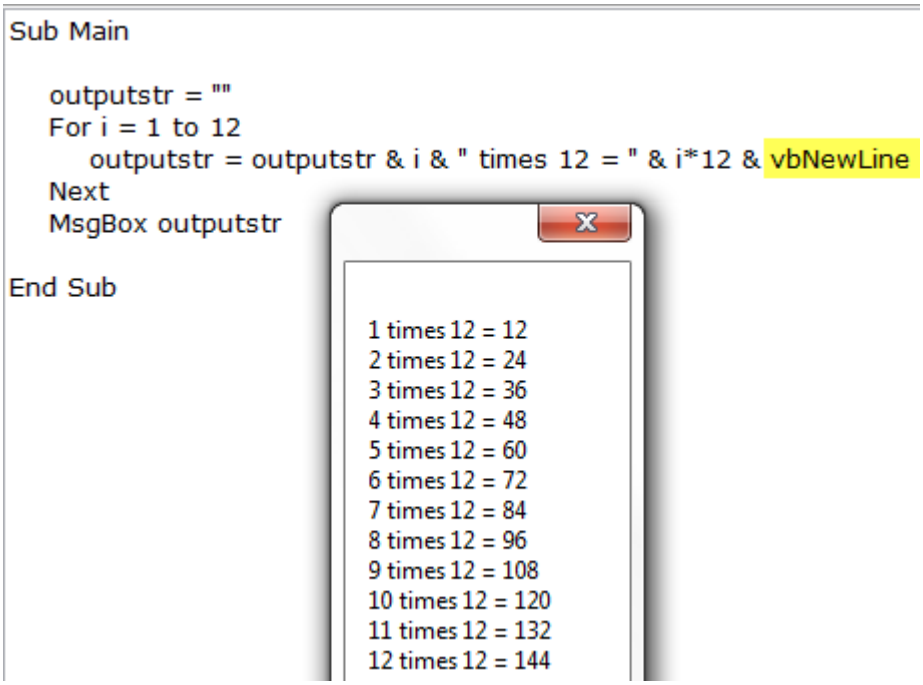
```
Sub Main

    val1 = 5
    val2 = 10

    MsgBox val1*val2      '' 50
    MsgBox val1/val2      '' 0.50
    MsgBox val1^2         '' 25
    MsgBox val2 Mod val1  '' 0

End Sub
```

Loops

For…Next

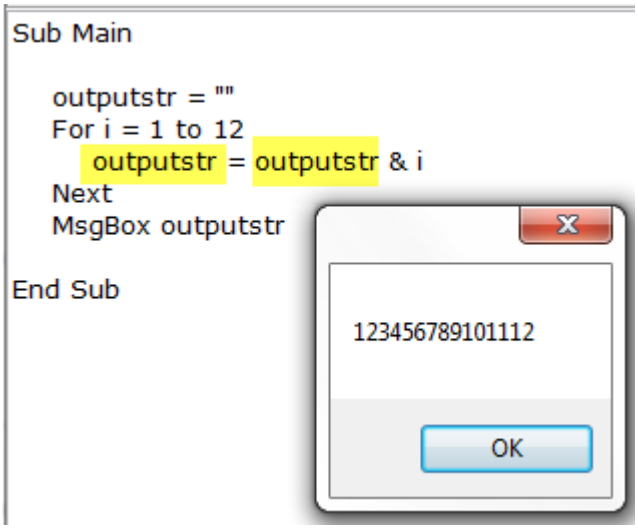
It is tedious having to always write one line per action. What if you need to do the same thing many times, but with a different value plugged in each time? A good example is a times table routine.

```
Sub Main

    outputstr = ""
    For i = 1 to 12
        outputstr = outputstr & i & " times 12 = " & i*12 & vbNewLine
    Next
    MsgBox outputstr

End Sub
```



Note the *vbNewLine*. This is a VB system constant, defined for you by Windows. The advantage of system constants is that if running on a different operating system such as Unix or OSx, which have a different line termination convention to Windows, then the operating system will enforce the correct terminator for you. Otherwise, you would need many slightly different versions of a script for different operating systems.

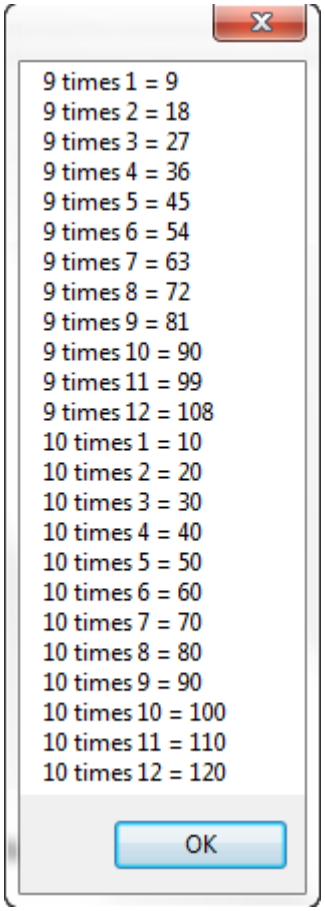
Note that you can assign a variable to itself.



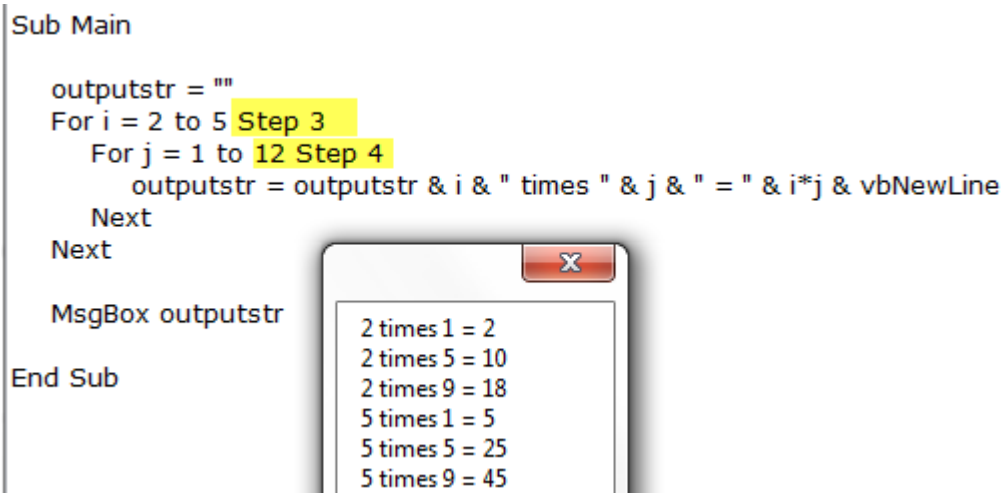
For…Next loops can be nested.

```
Sub Main

    outputstr = ""
    For i = 9 to 10
        For j = 1 to 12
            outputstr = outputstr & i & " times " & j & " = " & i*j & vbNewLine
        Next
    Next

    MsgBox outputstr

End Sub
```

The output is



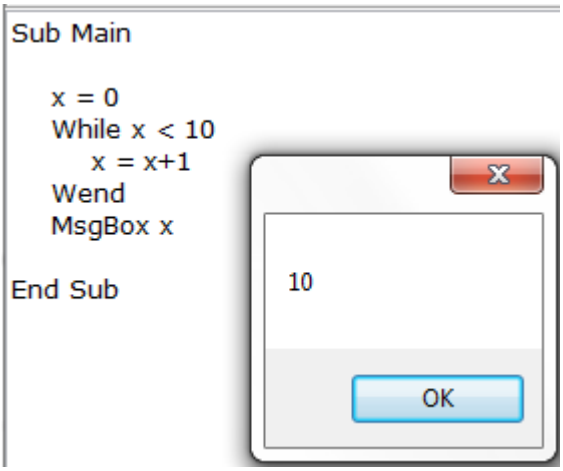
The loop iterations can be stepped.



While…Wend

While a condition is true, execute all steps until *Wend* is encountered.

```
Sub Main

    x = 0
    While x < 10
        x = x+1
    Wend
    MsgBox x

End Sub
```



When x reaches 10, the *While* loop breaks and x is no longer incremented.

Arrays

An array is a set of variables stored contiguously and accessed by index. For example, you frequently need to show a list of pets. You could do this:

```
Sub Main
    MsgBox "cat"
    MsgBox "dog"
    MsgBox "parrot"
    MsgBox "iguana"
End Sub
```

But that is tedious if required more than once. Instead, store the pet types in an array, as

```
Sub Main

    petarray = Array("cat", "dog", "parrot", "iguana")
    For i = 0 To 3
        MsgBox petarray(i)
    Next

End Sub
```

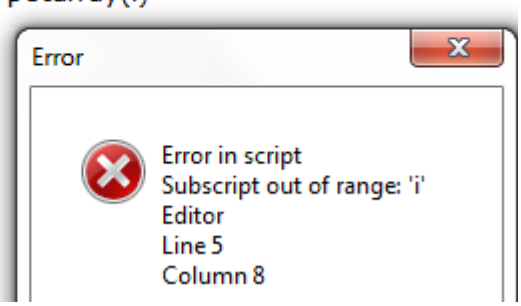
Petarray(0) is “cat” and petarray(3) is “iguana”.

Having to manually count the array items is error prone. If you accidentally do this

```
For i = 0 To 4
```

then you get an error, because there is no 5th position (at index 4).

```
Sub Main

   petarray = Array("cat", "dog", "parrot", "iguana")
   For i = 0 To 4
      MsgBox petarray(i)
   Next

End Sub
```



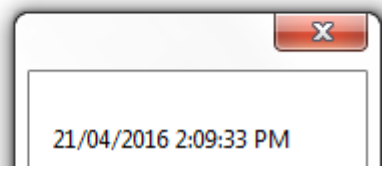
To guard against this, use the system function UBound() ('U'=upper), as

```
Sub Main

   petarray = Array("cat", "dog", "parrot", "iguana")
   For i = 0 To UBound(petarray)
      MsgBox petarray(i)
   Next

End Sub
```

If you do not know what values the array will hold, or want to populate the array dynamically, then you declare the array to be a particular size first.

```
Sub Main

    Dim timearray(3)
    For i = 0 To UBound(timearray)
        timearray(i) = Now
        MsgBox timearray(i)
    Next

End Sub
```

```
Sub Main

   Dim timearray(3)
   For i = 0 To UBound(timearray)
      timearray(i) = Now
      MsgBox timearray(i)
   Next

End Sub
```



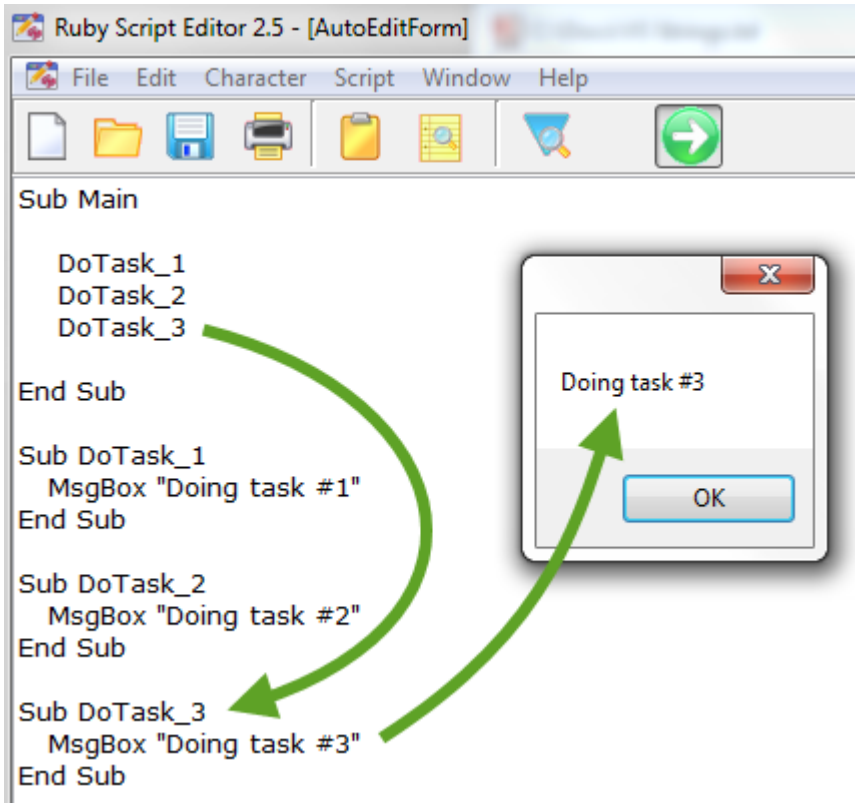
The array is declared using the *Dim* keyword. *Dim* is short for *Dimension* as a verb, ie, "Dimension an array called *timearray* with four positions, indexed as 0 to 3". Different langauges have different ways of declaring an array. The VB way is to use the Dim keyword.

Now is another system function which returns the current date and time.

Sub-routines

Sub is short for sub-routine. The Main routine (itself a subroutine) will typically call other sub-routines:

```
Sub Main
    DoTask_1
    DoTask_2
    DoTask_3
End Sub

Sub DoTask_1
   MsgBox "Doing task #1"
End Sub

Sub DoTask_2
   MsgBox "Doing task #2"
End Sub

Sub DoTask_3
   MsgBox "Doing task #3"
End Sub
```



The intention of subroutines is to break up long tasks into many sub-tasks.

MsgBox is itself a system sub-routine, provided out-of-the-box by the VB Script engine. How the string is displayed in a popup window is of no interest to us – just as long as it works.

Parameters

MsgBox has a string parameter. Similarly, we could initialise our tasks 1, 2 and 3 with some starter information by passing parameters, so that only one sub-routine is required. For example, if the tasks are to calculate something:

```
Sub Main

    Calculate "+", 1, 2
    Calculate "-", 3, 4
    Calculate "*", 5, 6

End Sub

Sub Calculate(operator, op1, op2)

    If      operator = "+" Then
        MsgBox op1 & operator & op2 & " = " & op1 + op2
    ElseIf operator = "-" Then
        MsgBox op1 & operator & op2 & " = " & op1 - op2
    ElseIf operator = "*" Then
        MsgBox op1 & operator & op2 & " = " & op1 * op2
    End If

End Sub
```

The parameters for the Calculate() sub-routine are an operator as a string and two operands as integers.

Functions

Functions are similar to sub-routines, in that they perform a task, but a function additionally returns a value. The function's return value is assigned to the function name.

```
Sub Main

    MsgBox Calculate("+", 1, 2)
    MsgBox Calculate("-", 3, 4)
    MsgBox Calculate("*", 5, 6)
    MsgBox Calculate("$", 5, 6)   '' $ is not handled by Calculate()

End Sub

Function Calculate(operator, op1, op2)

    If      operator = "+" Then
       Calculate = op1 + op2
    ElseIf operator = "-" Then
       Calculate = op1 - op2
    ElseIf operator = "*" Then
       Calculate = op1 * op2
    Else
        MsgBox "Unknown operator " & operator & " for Calculate()"
    End If

End Function
```

Note these highlights:

```
Sub Main

   MsgBox Calculate("+", 1, 2)
   MsgBox Calculate("-", 3, 4)
   MsgBox Calculate("*", 5, 6)
   MsgBox Calculate("$", 5, 6)  " $ is not handled by Calculate()

End Sub

Function Calculate(operator, op1, op2)

   If      operator = "+" Then
     Calculate = op1 + op2
   ElseIf operator = "-" Then
     Calculate = op1 - op2
   ElseIf operator = "*" Then
     Calculate = op1 * op2
   Else
      MsgBox "Unknown operator " & operator & " for Calculate()"
   End If

End Function
```

Function parameters must be parenthesised so that the VB parser knows to expect the expression to resolve to a single value.

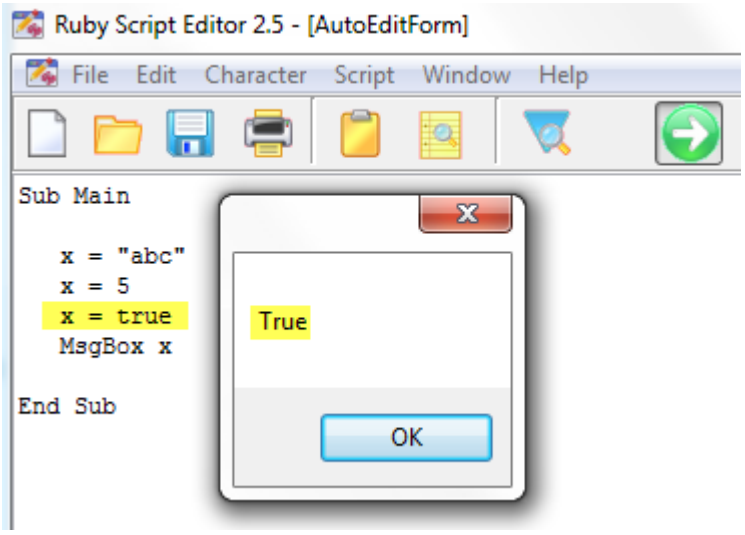
Calculate is a function, not a variable, but the assignment syntax is the same.

A function definition is terminated with *End Function*.

Variable Types

Numeric, String and Boolean

So far, we have seen numeric, string and Boolean variables. These are usually called primitive variables, because they contain only a single value. In VBScript, a variable type is determined only by context, and the type can change during execution.



Objects

Object variables, on the other hand, can contain multiple simultaneous instances of values, subroutines and functions (called *Properties* and *Methods* in object-speak). An object is typically organised as a hierarchy of parent/child sub-objects with the leaf nodes as the properties or methods. The members of the hierarchy are accessed by dot syntax. For example, to add a comment to a cell in Excel:

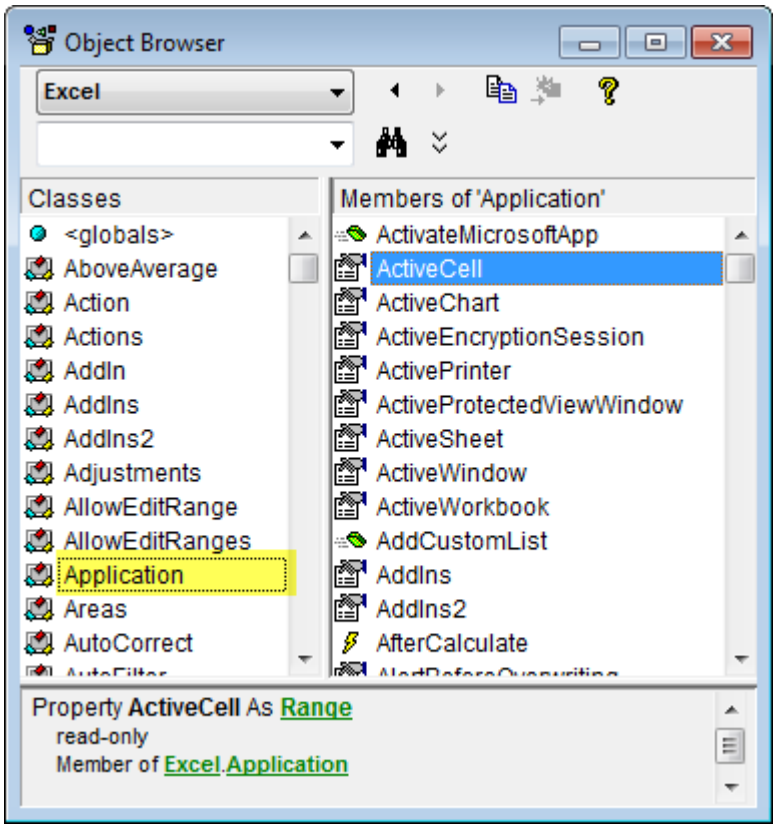
```
Application.ActiveCell.AddComment "This is a comment"
```

Application is the Excel application object.

ActiveCell is a child object of the Application object.

AddComment is a method (ie a subroutine) of *ActiveCell* which takes a string parameter.

The Application object members can be examined in Excel:



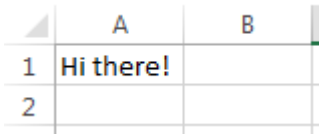
An object's hierarchy, methods and properties is called an *object model*.

Assigning an Object - Excel

An object variable is clearly a great deal more powerful than just a numeric or string variable, since it is the access point to the exposed functionality of complete applications, such as Excel, Word, or Ruby. To make it clear when you intend to assign an object, VBScript (and VBA) uses the *Set* keyword.

```
Sub Main

    Set exapp     = CreateObject("Excel.Application")    '' system magic to get Excel
    exapp.Visible = true                                 '' make visible
    Set exbook    = exapp.Workbooks.Add()                '' get a workbook from the app
    Set exsheet   = exbook.WorkSheets.Add()              '' get a sheet from the workbook
    exsheet.Name  = "Test"                               '' assign to the Name property
    exsheet.Cells(1,1).Value = "Hi there!"               '' put some text in cell A1
```

```
End Sub
```



Assigning an Object - Ruby

In the same way, we can get script access to Ruby's functionality like this:

```
Sub Main

    Set rub = CreateObject("Ruby.App1")               '' system magic
    Set rep = rub.GenTab("TestTab", "Gender", "ABA") '' assign table to rep
    rep.Save                                          '' invoke Save method

End Sub
```

The variable names rub and rep could be any legal name, eg

```
    Set mary = CreateObject("Ruby.App1")
    Set jack = mary.GenTab("TestTab", "Gender", "ABA")
    jack.Save
```

But this makes keeping track of what's going on rather difficult. You could be completely explicit, eg

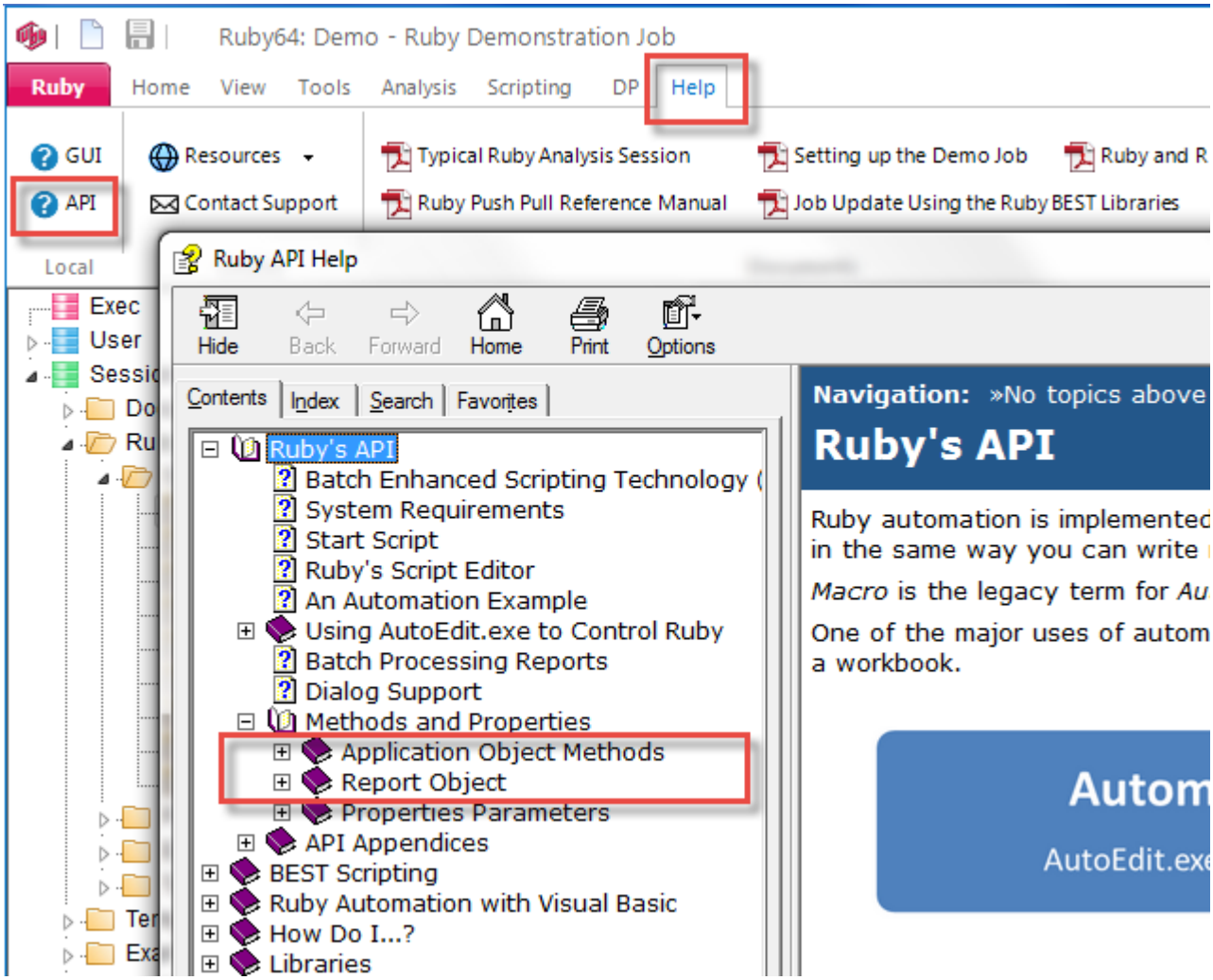
```
    Set ruby = CreateObject("Ruby.App1")
    Set report = ruby.GenTab("TestTab", "Gender", "ABA")
    report.Save
```

or

```
    Set table = ruby.GenTab("TestTab", "Gender", "ABA")
    table.Save
```

It is always good practice to use variable names which are short, but not too short at the expense of clarity.

The functionality exposed by the Ruby Application object and the Report object is fully documented here:



API is the TLA for *Application Programmer Interface*. Unlike Excel, which has hundreds, possibly thousands of objects, Ruby has only two: the application object and the report (document) object.

Generate Table, Copy to Excel

We can put the above Excel and Ruby examples together, to achieve something useful.

```
Sub Main

    Set rub = CreateObject("Ruby.App1")                    '' system magic to get Ruby

    Set exapp     = CreateObject("Excel.Application")      '' system magic to get Excel
    exapp.Visible = true                                   '' make visible
    Set exbook    = exapp.Workbooks.Add()                  '' get a workbook from the app
    Set exsheet   = exbook.WorkSheets.Add()                '' get a sheet from the workbook

    Set rep = rub.GenTab("Test Table", "Gender", "ABA") '' assign table object to rep
    rep.Copy "HTML,Labels"                                 '' put on clipboard, HTML format
    exsheet.Paste()                                        '' paste to sheet

End Sub
```

	A	B	C
1	Frequencies Column Percents	Male	Female
2	BrandX	4419	4442
3		89	89
4	BrandY	2083	2098
5		42	42
6	BrandZ	640	643
7		13	13

Examples

There are many examples of VBScript in the standard Ruby installation. See \Ruby\Jobs*.vbs for scripts which can be used on any job, and \Ruby\Jobs\Demo\Scripts*.vbs for job-specific scripts.

Documentation

The official documentation for VBScript is script56.chm, which can be downloaded from

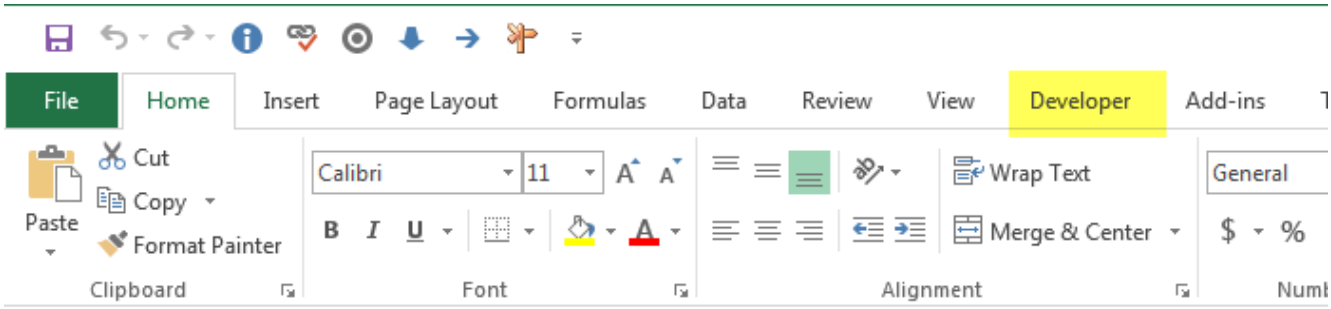
http://www.microsoft.com/en-au/download/details.aspx?id=2764

*.chm is an active file type, so you may have to unblock the file to access its contents, depending on your security settings.

Using Excel

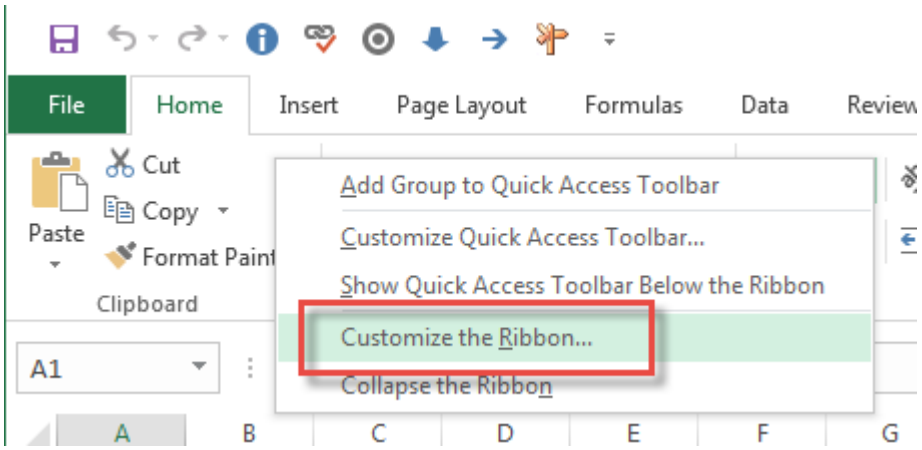
All of the above code examples can be executed in Excel, or any other Microsoft Office application.

First, confirm that your Excel has the Developer tab:

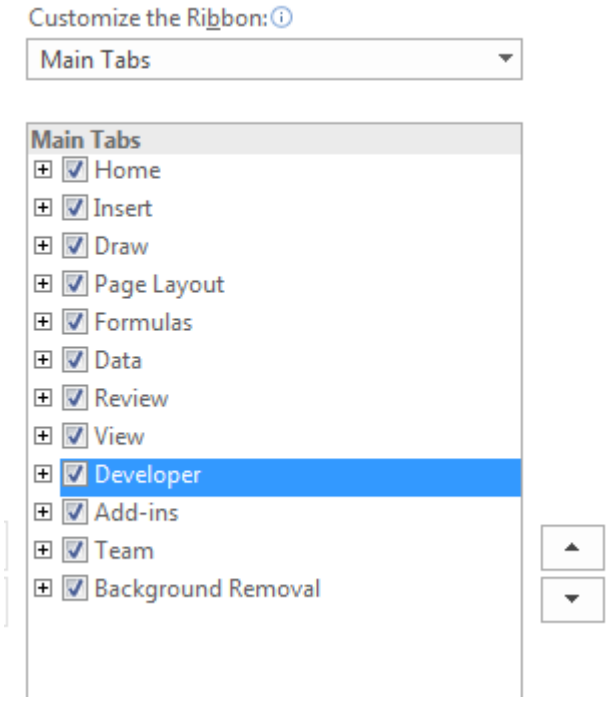


If not, then

- Right click on the Ribbon and select Customize Ribbon

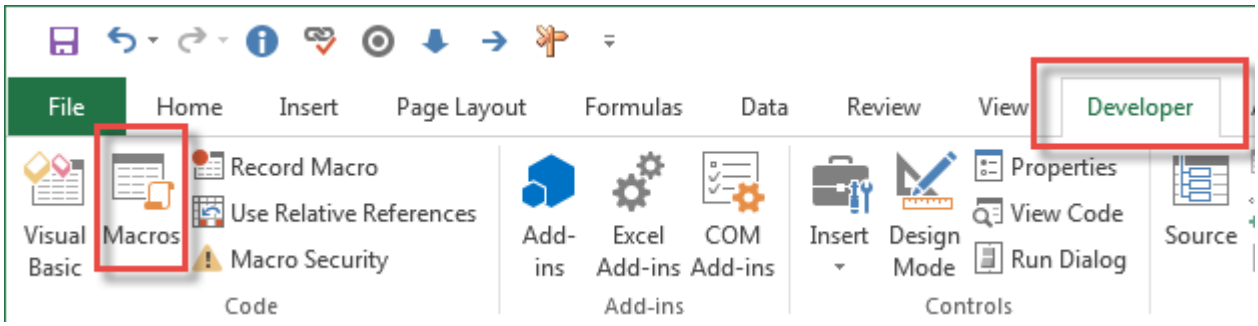


- Check the Developer item, OK



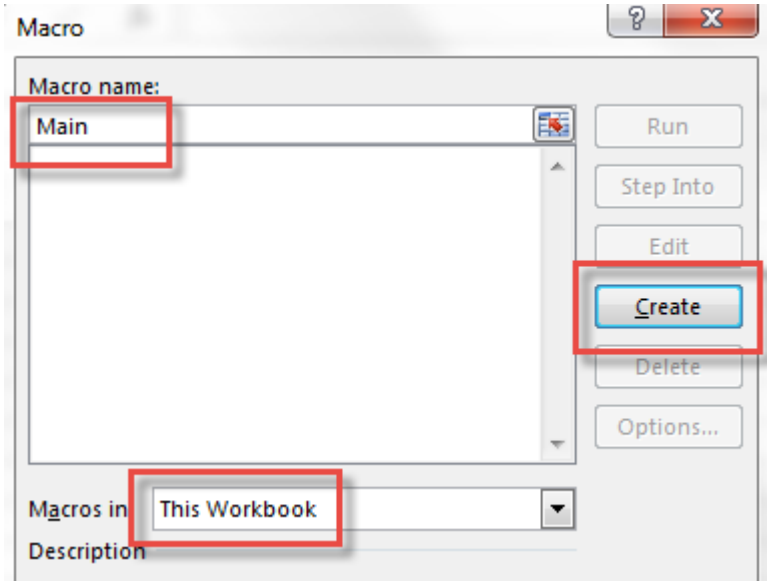
To enter a script

- Developer | Macros



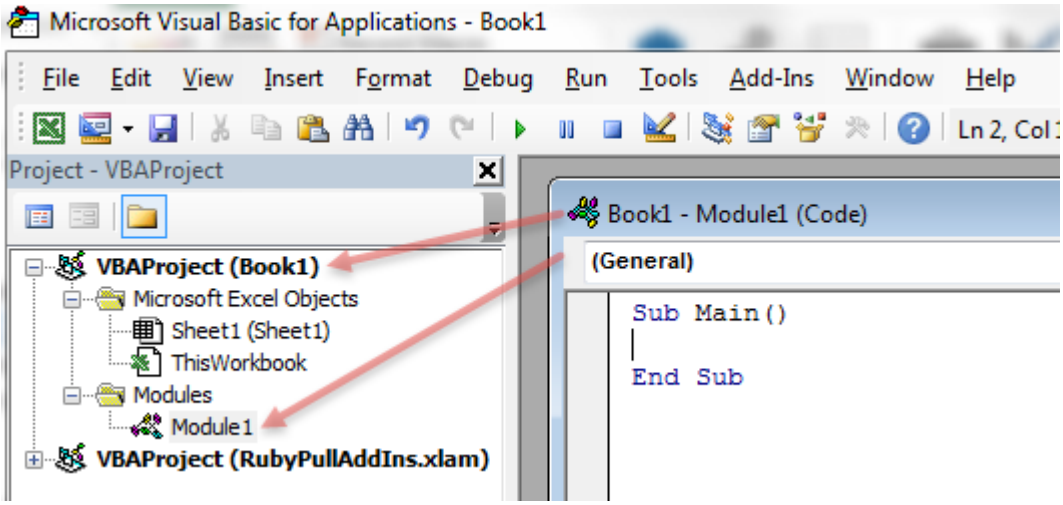
Then

- Enter the name as *Main*
- Select *This Workbook*
- Click Create

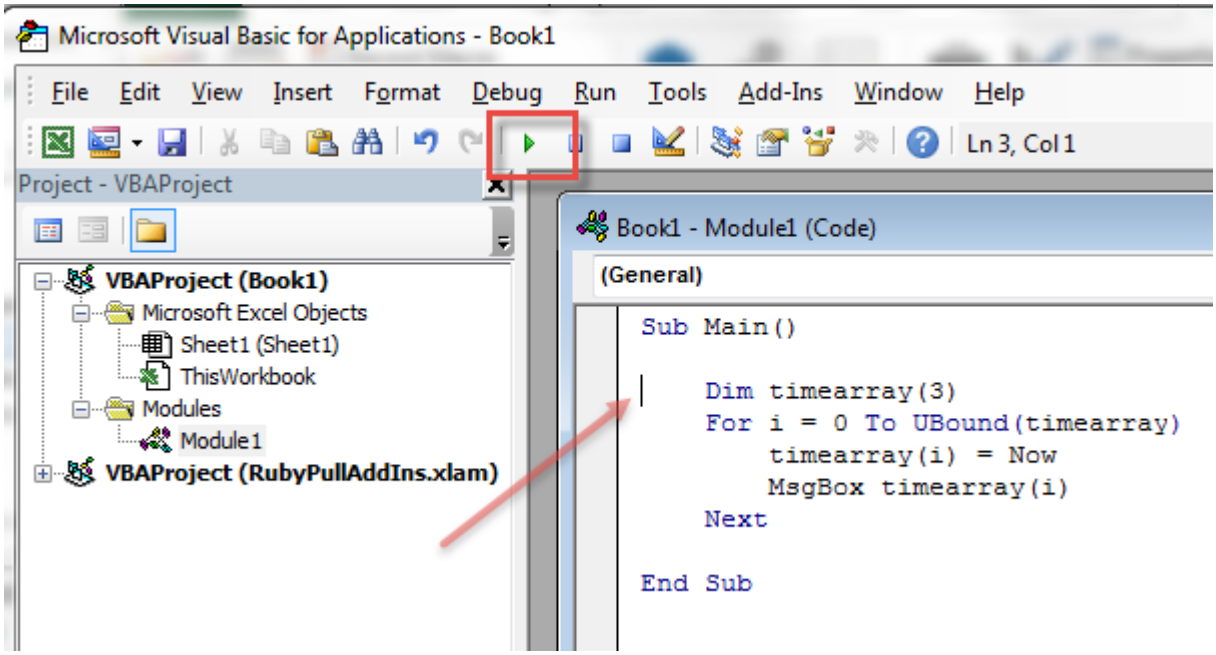


Excel allows you to use another name, but we retain *Main* to be consistent with the VBScript examples above.

A skeleton project is set up for you, as

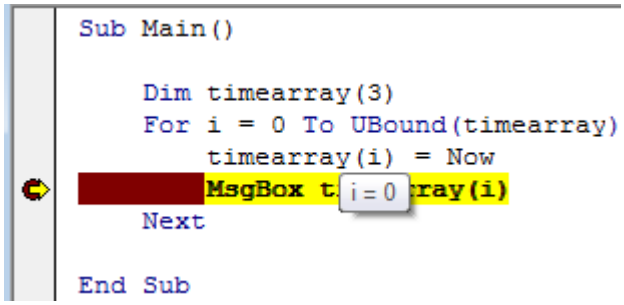


You can now copy/paste any of the above VBScript examples (except the Excel ones), eg



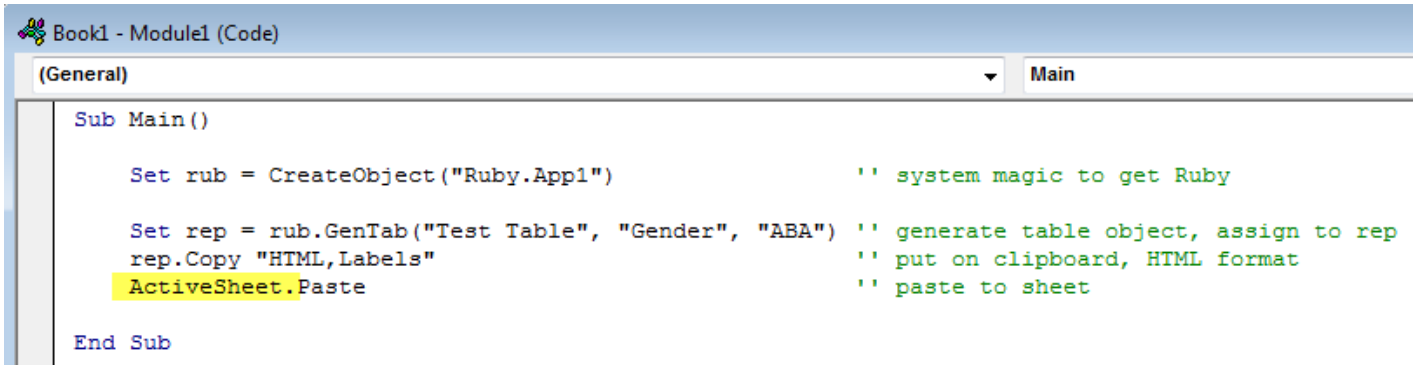
To run, put the cursor anywhere inside the Main subroutine body, and then click the Run button.

A major advantage of Excel over the built-in Ruby script editor is debugging. You can set breakpoints, watch variables and examine variable values by mouse hovering:



i = 0 tells me that I am on the first iteration of the loop.

The reason why the VBScript examples which call Excel cannot be used as-is in Excel is because you are already in Excel. The example which generates a Ruby table and copies it to Excel needs to be rewritten (much more simply), as



Other MS Office

Word, PowerPoint, Outlook, all use the same environment for scripting, but of course their object models are completely different.

Examples

For examples which are executed from Excel and call Ruby, see

\Ruby\Jobs\Demo\Docs\CustomDashboard.xlsm
\Ruby\Jobs\Demo\Docs\CustomSpreadsheetPull.xlsm
\Ruby\Jobs\Demo\Docs\GenTabSpecsServer.xlsm

Documentation

The official VBA documentation is presently at

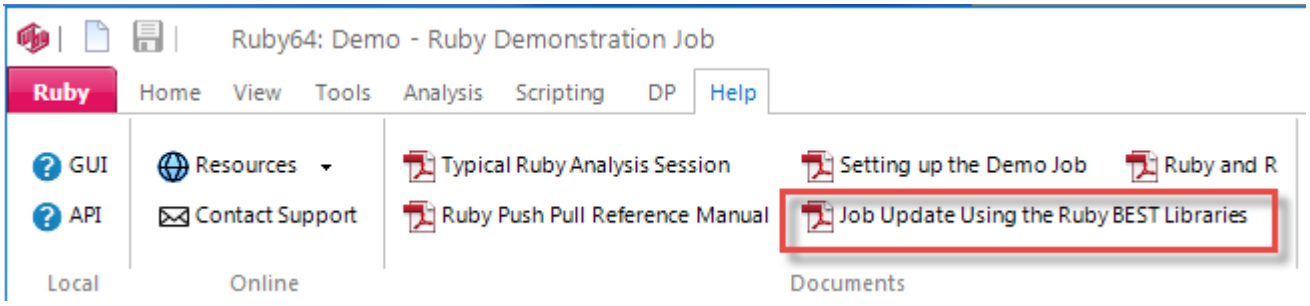
https://www.microsoft.com/en-us/download/details.aspx?id=9034

VB.NET

IDEs

VBScript (Ruby internal) is OK for simple tasks, but anything complicated benefits greatly from a complete and modern interactive development environment. You can use Visual Studio Express (free), Visual Studio Community (free to small enterprises), any commercial Visual Studio edition, or various free 3rd party .Net IDEs such as SharpDevelop. The screenshots below show SharpDevelop.

For Visual Studio Express download and installation, see page 9ff of *Job Update Using the Ruby BEST Libraries.pdf*



For SharpDevelop, see

http://www.redcentresoftware.com/sharpdevelop-an-alternative-ide-to-visual-studio-express/

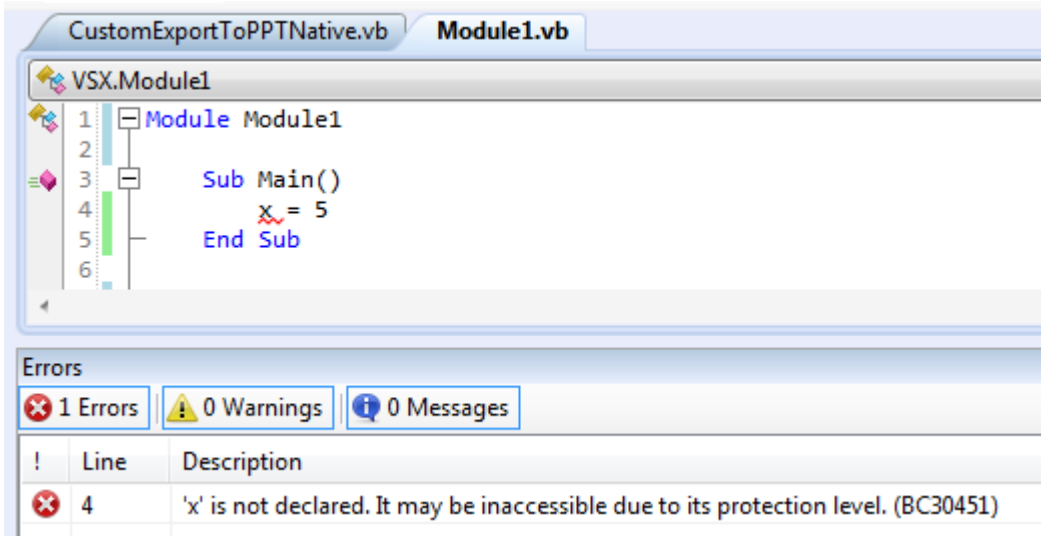
Syntax

Whereas VBScript and VBA are 99% identical languages, VB.Net has enough differences to render it a dialect. VB.Net is intended for use by software professionals, so it is stricter than VBScript – in particular you cannot just make up variables as you go along – they have to be declared before use.

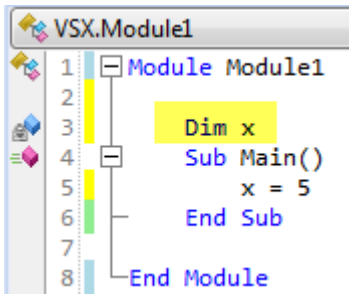
For example, this VBScript

```
Sub Main
    x = 5
End Sub
```

gives an error:

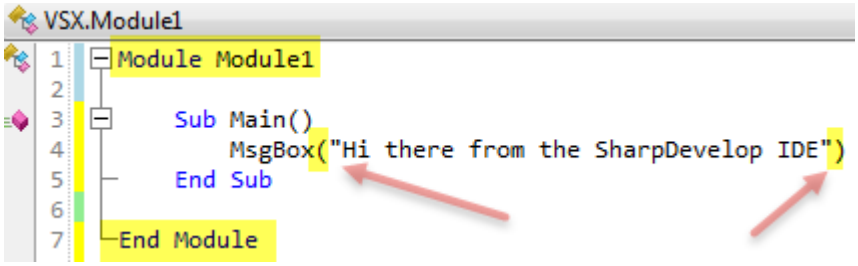


The fix is to declare x first, as

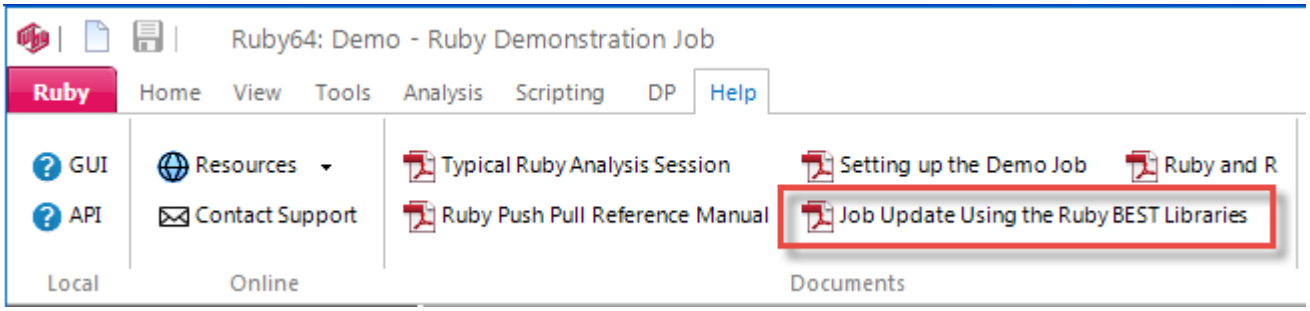


It would have been better IMHO if the designers of the original BASIC language had used *Dec* for *Declare* rather than *Dim* for *Dimension*, but it is as it is, so Dim means to Declare, that is, you say up front what variables you need.

Some other syntactic differences are: each file must have an internal Module name, and all subroutine calls need parentheses around parameters.



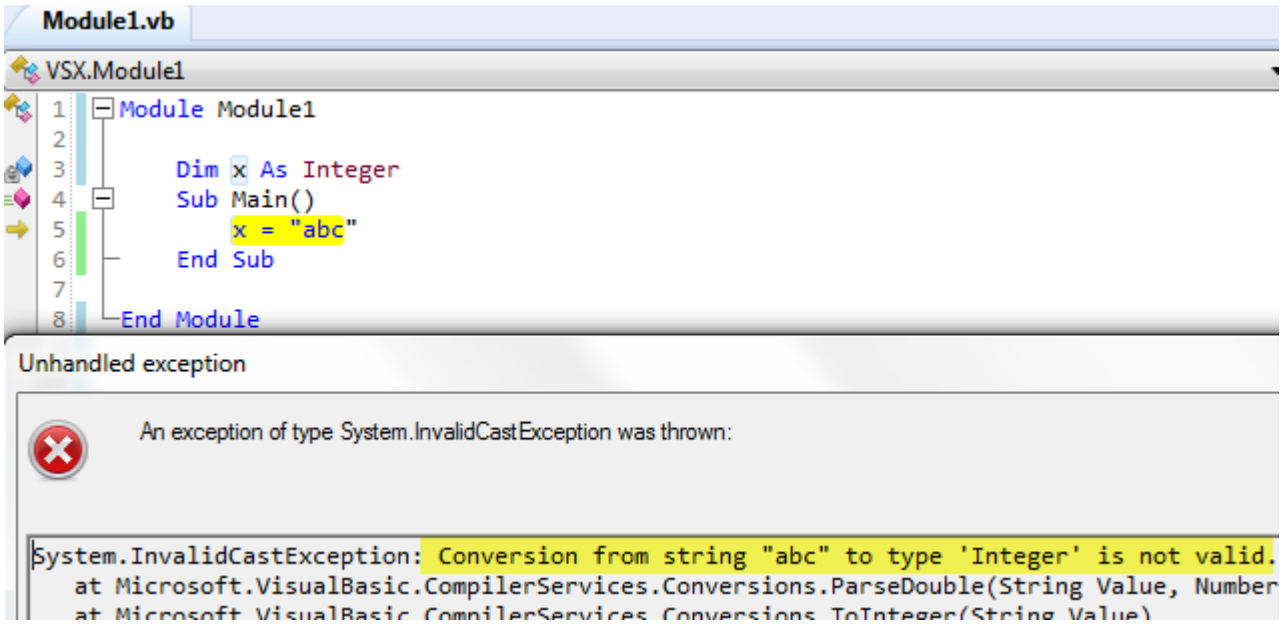
The default module name is *Module1*, but you can change it to any legal name. For details on other syntax differences, see page 85ff of *Job Update Using the Ruby BEST Libraries.pdf*:



Variable Typing

Whereas VBScript tries to convert between strings and numbers or other types automatically, the .Net environment can help catch design-time bugs and improve performance by typing variables as integer, string, boolean, double, etc.

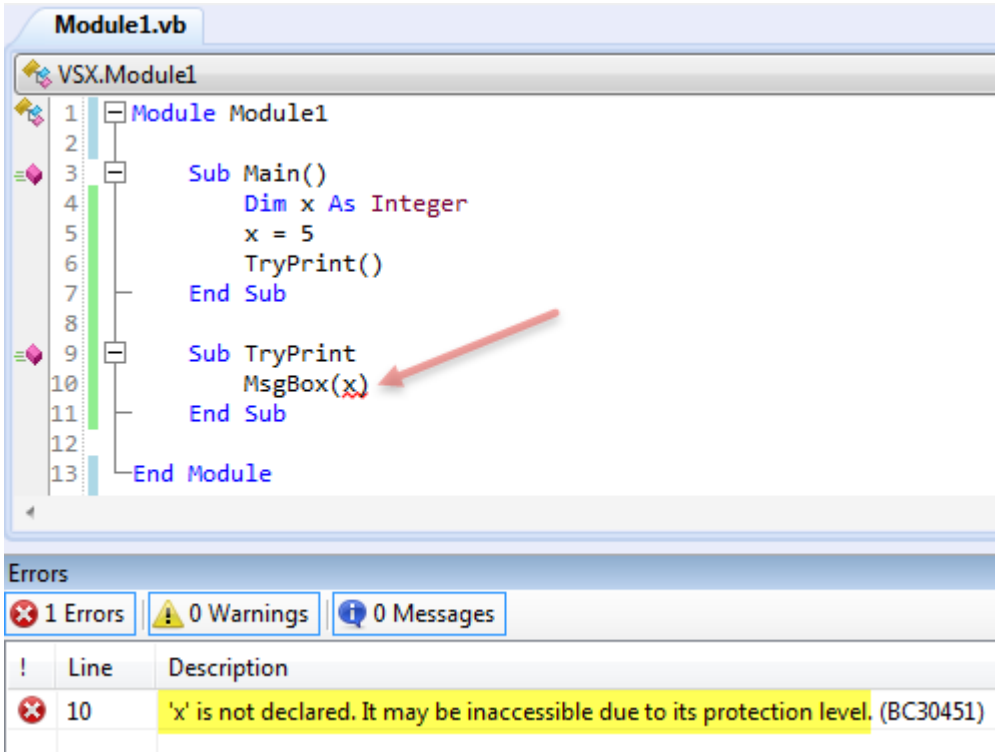
This gives an error because x is declared as an integer:



If there is no type indicated in the declaration, then the variable is a *Variant*, which means it can be assigned any type.

Scope

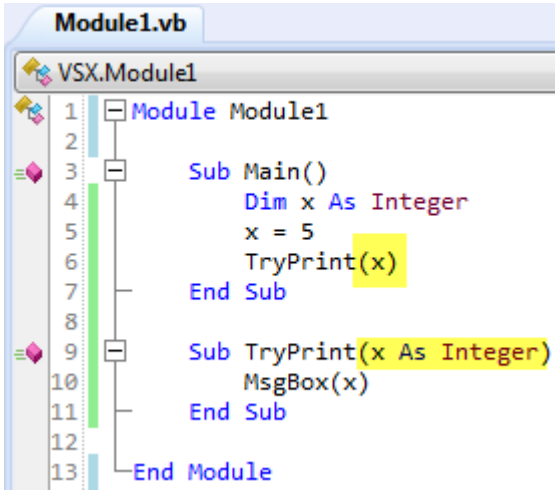
If a variable is declared outside of all subroutines, then it is available to all (as immediately above), but if within a subroutine, then only to that subroutine.



The variable x is declared inside *Main()*, so it is not available to *TryPrint()*.

Parameter Types

Passed parameters need to be typed also. To avoid the above error, x needs to be passed to TryPrint(), as



Object Types

In VB.Net, objects are strongly typed, and each object has its own type. This prevents accidental mix-ups, by ensuring an object is treated in the right way. Here is the VBscript which generates and copy/pastes a table to Excel translated to VB.Net:

```
Imports Ruby
Imports Excel = Microsoft.Office.Interop.Excel
```

```
Module Module1

    Sub Main()

        Dim rub As Ruby.App1 = New Ruby.App1()
        Dim rep As Ruby.RubyReport = New Ruby.RubyReport()

        Dim exapp As Excel.Application = New Excel.Application()
        exapp.Visible = true
        Dim exbook As Excel.Workbook = exapp.Workbooks.Add()
        Dim exsheet As Excel.Worksheet = exbook.WorkSheets.Add()

        rep = rub.GenTab("Test Table", "Gender", "ABA")
        rep.Copy("HTML,Labels")
        exsheet.Paste()

    End Sub

End Module
```

This is clearly a bit more complicated.

The two *Imports* lines get system access to the two objects. *Microsoft.Office.Interop.Excel* is rather long, so that is assigned to the much shorter *Excel* (or any other legal name of your choosing).

The Ruby object type is *Ruby.App1*, and the report (or document) type is *Ruby.RubyReport*.

The next four lines start up Excel, make it visible, and add a workbook and worksheet.

Note that objects must be *New*'ed, and that the parameter string to *rep.Copy* must be parenthesised.

The *Set* keyword, required in VBScript and VBA, is not part of VB.Net.

Examples

There are many examples in the standard Ruby installation. Look for folders called \VSX in the Demo job, and in the jobs under the \Ruby\Jobs\BEST\ subdirectory.

Documentation

The official documentation is at

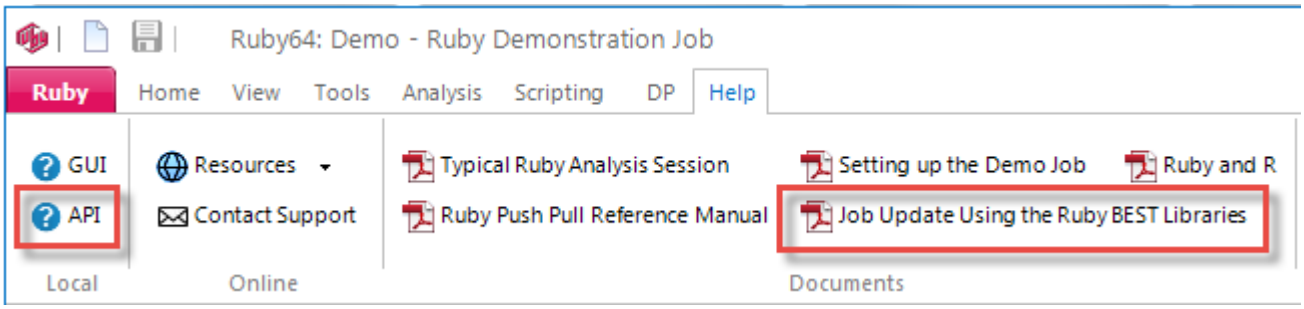
https://msdn.microsoft.com/en-us/library/2x7h1hfk.aspx

DotNet Pearls has many clear and simple examples, see

http://www.dotnetperls.com/vb

RUBY DOCUMENTATION

The major docments for Ruby are the API Help and *Job Update Using the Ruby BEST Libraries.pdf*:



[end of document]