VB6 To Tcl mini–HOWTO

Mark Hubbard

Digital Connections Inc.

<markh@dcisite.com>

Pradeep Padala – Conversion from HTML to DocBook v4.1.

Revision History

Revision 1.0	2003–04–30	Revised by: tab
Initial release, reviewed by LDP		
Revision 0.9	2003–04–08	Revised by: ppadala
Docbook conversion		
Revision 0.8	2002–07–08	Revised by: mark
Original Document		

A 15 Minute Tcl Tour For Visual Basic and VBScript Programmers

Table of Contents

1. Introduction

VB and VBScript programmers: I know how you feel. Really. As a Microsoft Certified Professional in VB6, I've been doing those languages for 7 years. I really liked them, until I got over the hump in Tcl and started noticing the differences in flexibility that are shown here. If Tcl looks completely alien to you, and you wonder how in the world they dreamed it up, hold it up beside a piece of C code, or a UNIX shell script. I think those are what influenced it the most. UNIX shell scripts are a lot more advanced than MS Windows shell scripts, even those on NT/2000. In fact, UNIX shell scripts have a lot of the capabilities shown here. Both Tcl and shell script are based largely on string substitution. I chose to study Tcl over shell scripts because Tcl code is much more verbose and English−like (and therefore maintainable) than shell scripts, which tend to be cryptic. Some of the shell script command names are just punctuation alone!

Tcl also runs easily on the "big 4" PC platforms (Linux, *nix, Windows, Mac) as well as some others. This is promised by Java(tm), but delivered just as much (or more) by Tcl. And unlike Java and VB, Tcl is free of any commercial influences (which is true freedom, not just "free of charge"); over the years its development path sticks closer to what is really needed and wanted by you, its developers and potential developers. There has been no parent company to steer Tcl away from that and toward the company's own interests. The most startling contrast of all between Tcl and VB is that Tcl may even overshadow all the technical differences shown below.

---

2. Examples

Table 1. Differences

<table>
<tr><th>VB6</th><th>Tcl/Tk 8.3</th></tr>
<tr><td colspan="2">Notes/differences</td></tr>
<tr><td>dim a as integer
dim b as integer
a=1 : b=0</td><td>set a 1; set b 0</td></tr>
<tr><td colspan="2">Separator for multiple commands per line. Tcl uses a semicolon. Multiple commands per line is generally considered bad form, but the semicolon is also used to implement partial−line comments, so it is illustrated here.</td></tr>
<tr><td>' this is a whole line</td><td># this is a whole line</td></tr>
<tr><td colspan="2">Full−line comment. Neither language requires a space after the comment marker.</td></tr>
<tr><td>dim a as integer
a=1 'this is a partial−line comment</td><td>set a 1 ;# this is a partial−line comment</td></tr>
<tr><td colspan="2">Partial−line comment. Note the semicolon, used as if the comment is another command on that line.</td></tr>
<tr><td>dim s as string
s="/data/docs/vb6_to_tcl.htm"</td><td>set s {/data/docs/vb6_to_tcl.htm}</td></tr>
<tr><td colspan="2">Assignment of a string quoted with braces. Most Tcl substitutions are NOT done in a string quoted with braces. If the string contains variables or other items that would be substituted, these will be deferred, but may be substituted at a later time. This is often done by commands that implement control structures, such as 'if' or 'while'. Once you start to get familiar with Tcl initially, try to thoroughly understand this process because it's important to getting 'good' in Tcl.</td></tr>
<tr><td>(No equivalent)</td><td>set s "/data/docs/vb6_to_tcl.htm"</td></tr>
<tr><td colspan="2">Assignment of a quoted string. All Tcl substitutions (variables, commands, backslashes) are available within a quoted string.</td></tr>
<tr><td>(No equivalent)</td><td>set s /data/docs/vb6_to_tcl.htm</td></tr>
<tr><td colspan="2">Assignment of an unquoted string. All Tcl substitutions (variables, commands, backslashes) are available within an unquoted string. The interpreter simply takes the string as the third word in the set command (second argument to the set command). This works if there is no whitespace or certain other characters in the string. Use judiciously, especially when dealing with arbitrary data entered by the user.</td></tr>
<tr><td>dim s as string
s = vbCrLf &"Free software is not just" &vbCrLf _
  &"about being 'free of charge'" &vbCrLf _
  &"but about freedom to create" &vbCrLf _
  &"and use the best possible tools." &vbCrLf</td><td>set s {
Free software is not just
about being 'free of charge'
but about freedom to create
and use the best possible tools.
}</td></tr>
<tr><td colspan="2">Assignment of multi−line string. Note the more cluttered syntax in VB, which makes it more difficult to read than the Tcl code.</td></tr>
<tr><td></td><td></td></tr>
</table>

```
dim s as string
dim t as string
s = trim(t)
```

```
set s [string trim $t]
```

Assignment of function return value. The third word of this set command is surrounded in square brackets. That means it is itself a command to be executed, with the result taking its place as the third word of the set command.

```
dim s as string
dim t as string
s = lcase(trim(t))
```

```
set s [string tolower [string trim $t]]
```

Assignment of function−of−function.

```
dim x as double
dim y as double
x = (y + 10) * 5
```

```
set x [expr {($y + 10) * 5}]
```

Assignment of result of a mathematical expression. The Tcl interpreter relies on the expr command to evaluate mathematical or logical expressions. Many other commands such as 'if' or 'while' also rely on expr in their implementation. When used explicitly, expr should be passed a single argument which is a string containing the expression (as shown here). That could get cumbersome in simple cases where you just want to add a certain increment to a variable. Try using the incr command for that instead.

```
dim s as string
s = s &"more text"
```

```
append s {more text}
```

Append to an existing string. This is one of the slowest operations in VB, but is typically very speedy in Tcl. Speed is important here because it is often done within loops or compound loops.

```
dim s as string
dim t as string
dim u as string
s = "I'll ask " & t & " to email " & trim(u) & " with the price"
```

```
set s "I'll ask $t to email [string trim $u] with the price"
```

Building a string by substitution.

```
print "hello"
```

Displays hello.

Print to console (VB actually prints to a form or to the debug window).

```
sub my_sub (byval a as integer, byval b as string)

    debug.print "I'll ask " & b

end sub

function my_function (byval a as integer, _
        optional byval b as string = "Mark") _
        as string

    my_function = "I'll ask " & b

end function
```

```
proc my_sub {a b} {

    puts "I'll ask $b"

}

proc my_function {a {b Mark}} {

    return "I'll ask $b"

}
```

Procedure definition. Note that VB uses a separate syntax for subs and for functions. Tcl uses the **proc** command to define either one. **proc** itself is an ordinary Tcl command that executes like any other command. Its first argument is a Tcl list of the parameters of the new procedure. Its second argument is a large string containing the body of the new procedure (actual Tcl script). *Important:* Tcl is case sensitive in almost all operations, including all references to command names and variable names, as well as (by default) string data comparisons. So a call to **Proc** would cause an error (capital P), as would a call to `My_Sub`, or reference to the variable `B` within `my_sub` (b was defined as lower case).

```
dim i as integer
if i < 0 then i = 0 else i = i - 1
```

```
if {$i < 0} {set i 0} {incr i -1}

# alternate form
if {$i < 0} then {set i 0} else {incr i -1}

# another alternate form
if {$i < 0} then {
    set i 0
} else {
    incr i -1
}
```

'if' conditional execution. The Tcl 'if' command ignores the optional keywords 'then' and 'else' if they are present. Since both code blocks are just strings, they can be enclosed in braces and nicely formatted as shown. To avoid syntax errors, also enclose any non−trivial test expression in braces. That way substitutions (such as $i here) are deferred until the 'if' command passes the test expression to the expression parser.

```
dim i as integer
i = 1
while i < 2000
    i = i * 2
wend

'alternate form
i = 1
do while i < 2000
    i = i * 2
loop
```

```
set i 1
while {$i < 2000} {
    set i [expr {$i * 2}]
}
```

'while' loop. This is similar to the Tcl 'if' command in that it takes a test expression as its first argument, followed by a string of code.

```
dim i as integer
for i = 0 to 8
    'nine passes 0-8
    debug.print i
next
```

```
for {set i 0} {$i < 9} {incr i} {
    # nine passes 0-8
    puts $i
}

# alternate form
for {set i 0} {$i <= 8} {incr i} {
    # again, nine passes 0-8
    puts $i
}

# another alternate form
for {set i 1} {$i <= 9} {incr i} {
    # nine passes 1-9
    puts $i
}

# yet another alternate form - less readable
set i 1
for {} {[incr i] <= 9} {} {
    # nine passes 1-9
    puts $i
}
```

'for' loop with an integer counter. In Tcl (or any other language) this is equivalent to a 'while' loop. In some languages such as VB, 'for' is not as flexible as 'while'. In Tcl this is not the case. Anything can be used as the initialization code, the test−for−continuation expression, and the increment code. Those pieces are not restricted to doing anything in particular, as you can see by the final example.

```
dim c as new collection
dim o as object
c.add "Mark"
c.add "Roy"
c.add "Brian"
for each o in c
    debug.print o
next
```

```
set c [list Mark Roy Brian]
foreach o $c {
    puts $o
}
```

Loop through items in a data structure. In Tcl, a list data structure is used. VB has no direct equivalent to that, but a collection object is the most similar. Note that VB collections are far slower than Tcl lists in typical operations due to the overhead of using method calls to objects. Also note that there are *far more powerful and creative uses* of the foreach command that are not shown here. Those have no direct equivalent in VB.

```
dim s as string
select case s
    case "John"
        debug.print "Mellencamp"
    case "Steve"
        debug.print "Tyler"
    case else
        debug.print "Unknown"
end select
```

```
switch -exact $s {
    John {puts Mellencamp}
    Steve {puts Tyler}
    default {puts Unknown}
}
```

One−of−many execution. Note the Tcl version is case sensitive. In VB it often is not, depending on the 'option compare' that is in effect for the module. The `-exact` option specifies an exact string match is required, as opposed to a pattern match or regular expression match (this has no bearing on case sensitivity). Also note that there are more powerful and creative uses of the switch command that are not shown here.

```
on error goto handler
debug.print a 'a is undeclared.
...
handler:
debug.print err.number, err.description
```

```
if [catch {
    puts $a ;# a has not been set
} my_err] {
    puts "error message: $my_err"
    puts "stack trace: $errorInfo"
    # these things would have been shown
    # by the default error handler anyway.
} else {
    puts {All is well.}
    # the else block is optional.
}
```

Error handling. In VB, handling errors concisely can be a problem, especially if different actions need to be taken based on which part of the code failed. Tcl **catch** command neatly solves these problems. In addition, Tcl automatically provides a stack trace of the code that failed. In VB, the stack trace has to be explicitly built by the code, if a stack trace is desired while the application is in production (not in the IDE). This is an advantage for Tcl when debugging in the field. Note that **catch** returns a boolean 1 or 0, which is typically used with 'if', as shown here.

```
(No equivalent)
```

```
set i [expr $e]
```

Pass an arbitrary mathematical expression to the interpreter for evaluation. This could be an expression entered by the user, or composed by earlier code. This is one of the most powerful aspects of Tcl. It is not available at all in VB.

```
(No equivalent)
```

```
set s [eval $c]
```

Pass arbitrary code to the interpreter for execution. This could be some script entered by the user, or composed by earlier code. This is one of the most powerful aspects of Tcl. It is not available at all in VB.

```
(No equivalent)
```

```
source my_script.tcl
```

Pass an arbitrary filename to the interpreter for execution of that file as a script. This is one of the most powerful aspects of Tcl. It is not available at all in VB.

<table>
<tr><td>

```
(No equivalent)
```

</td><td>

```
set var_name marks_age
incr $var_name
```

</td></tr>
<tr><td colspan="2">Perform operations on an arbitrarily−chosen variable. The code shown here will increment the variable <code>marks_age</code>. Its name (the string "marks_age") is stored in the variable <code>var_name</code>. In fact, all parts of every command are subject to one pass of substitution by the interpreter just prior to execution. So any part of any command (even the name of the command itself) can be varied based on data or any other criteria. This is one of the most powerful aspects of Tcl. It is not available at all in VB.</td></tr>
<tr><td>

```
dim s as string
dim li as string
dim f_num as integer
s = ""
f_num = freefile
open "my_file.txt" for input as #f_num
while not eof(f_num)
    line input #f_num, li
    s = s & li & vbCrLf
wend
close #f_num
```

</td><td>

```
set f [open my_file.txt r]
set s [read $f]
close $f
```

</td></tr>
<tr><td colspan="2">Read whole file into a variable. This VB code is very slow for even moderately large files. And it has no way to deal with newline characters in the data. The Tcl code accepts and preserves newlines in the data. It also normalizes different newline characters into a single kind of standardized newline character (by default). This code applies equally well to raw data, or Tcl lists, or Tcl arrays. The <code>r</code> in the open command indicates 'read' mode.</td></tr>
<tr><td>

```
dim a(1 to 3) as string
a(1) = "Mark"
a(2) = "Brian"
a(3) = "Roy"
'oops - need more elements
redim preserve a(1 to 10) as string
a(4) = "John"
```

</td><td>

```
array set a [list 1 Mark 2 Brian 3 Roy]
set a(4) John
# now some different kinds of
# element names in the same array
set a(Red) Hat
set a(Linux,RedHat) 7.1
```

</td></tr>
<tr><td colspan="2">Array vs. Array. VB arrays are restricted to using numbers as subscripts (subscripts, or indexes, are called 'element names' in Tcl). And the array must be declared to be a certain size − expanding it requires a (slow) 'ReDim Preserve' operation. Tcl arrays automatically expand, and they use a super−efficient hash table implementation to handle even hundreds of thousands of elements with superior speed. Tcl uses any kind of data for an element name, and different styles can even be mixed within the same array. There are no restrictions on the number of dimensions in each element. Tcl provides simple ways to iterate through the array, or through only certain elements in the array (by filter). You can also obtain a full or partial list of the element names, and do other operations more conveniently than in VB. To get just a portion of those capabilities in VB requires the use of a collection or dictionary object. Each of those comes with its own quirks and pitfalls, such as even higher overhead than a VB array.</td></tr>
<tr><td>

```
(No equivalent)
```

</td><td>

```
array set my_array $my_list
set my_list [array get my_array]
```

</td></tr>
<tr><td colspan="2">List to array, and back. Easy and rapid translation between these two primary data structures means that the tools for each one can be applied to both. They multiply each other's usefulness.</td></tr>
<tr><td>

```
dim a(1 to 100) as string
dim i as integer
dim f_num as integer
f_num = freefile
open "my_file.txt" for output as #f_num
for i=1 to 100
    print #f_num, a(i)
next
close #f_num
```

</td><td>

```
set f [open my_file.txt w]
puts $f [array get a]
close $f
```

</td></tr>
<tr><td colspan="2">Write whole array. In this VB code, and frequently in other VB code, newlines and possibly other characters appearing in the data will cause errors during a later step (the read−back). This becomes a problem whenever your code deals with arbitrary data entered by the user. In Tcl they do not − the data is kept "clean" at all times. In addition, various combinations of carriage return (0x0D or decimal 13) and line−feed (0x0A or decimal 10) characters are automatically normalized by default. Note that these two examples don't produce identical output files. The Tcl example, like the VB, writes a plain text file. But the Tcl file will</td></tr>
</table>

be read back in (by Tcl) and automatically have the same number of elements, same element names, etc.. The Tcl list data structure is used for this. Using it ensures that the data is formatted in a concise, non−ambiguous, textual representation. It is also readable and writable by humans.

```
(No equivalent)
```

```
set f [open my_file.txt w]
puts $f [array get a red*]
close $f
```

Write certain elements of an array. In the VB, a collection or dictionary object would have to be used for this. A loop would iterate through all the elements and select them as appropriate. In the Tcl, the array's name is `a` and a string pattern of `red*` (case sensitive) is used as a filter to select elements at high speed.

```
(No equivalent)
```

```
set my_list [lsort $my_list]
```

Sort a list. The sort can be reversed, or ordered by numeric value, etc. It can also order a list of sublists using an index element. Tcl contains a full suite of commands for manipulating the list data structure. See also **lappend**, **linsert**, **lreplace**, **lsearch**, **concat**, **split**, **join**, etc. Tcl lists can also be nested arbitrarily, and the **foreach** command has no trouble dealing with that.

```
' requires a reference to ADO
' assume we have a connection called conn
dim rs as new recordset
rs.open "select id, name, age from people", _
    my_connection, adOpenStatic
' processing code goes here
rs.close
set rs=nothing
```

```
package require tclodbc
# assume we have a connection called conn
conn read a "select id, name, age from people"
# processing code goes here
unset a ;# get rid of this array
```

Retrieve a simple array of data from a database table. In VB data is always retrieved in a recordset object. In Tcl it can be read into an array and/or a list, depending on your needs, and the database package in use.

```
(No equivalent)
```

```
package require http
set httpTrans [http::geturl $pageURL]
upvar #0 $httpTrans state
if {$state(status) == {ok}} {
        puts $state(body)
}
```

Retrieve a document or file from a web server.

```
(No equivalent)
```

```
regexp −all {src=['"](.+?)['"]} $body my_images
```

Complex string pattern search and extraction. Tcl uses *regular expressions* for this. *Regular expression* is a specification for a string pattern to be matched, similar in concept to the wildcard patterns used with VB's 'like' operator, except on steroids − a *whole lot* of steroids. Regular expressions are several times more powerful and flexible than 'like' patterns. For an informal introduction to regular expressions, see http://zez.org/article/articleprint/11. Tcl's regular expression parser is written in hand−optimized C code and is available to Tcl in several different commands (**regexp**, **regsub**, **lsearch**, etc). The simpler, less powerful versions you're used to are also available for use in several different commands (**glob**, **string match**, **lsearch**, and so on). This example would take 15 to 50 lines of VB code, depending on how robust and how tolerant of different situations it needs to be. In addition, that is some of the most difficult, error−prone, and slowest code that can be written in VB (voice of experience). Here, the code quickly obtains a list of the URLs of every image on an HTML page.

```
(No equivalent)
```

```
set find {<tr>(.*?)<td>(.*?)</td><td>(.*?)</td><td>(.*?)</td>(.*?)</tr>}
set replace {<tr>\1<td width=20%>\2</td><td width=40%>\3</td><td width=30%>\4</td>\5</tr>}
regsub −all −nocase $exp $body $replace result
puts $result
```

Complex string pattern search and substitution. Again, Tcl uses regular expressions. This example would take 40 lines of VB code or more, especially if it is logically organized with sufficient comments for a maintenance programmer to follow it. And again, it is some of the most difficult, error−prone, and slowest code that can be written in VB. Here the set of three cells in *every row in the HTML body* is altered systematically, while the contents of each cell is preserved.

```
(No equivalent)
```

```
set handle [socket markhpc.dcisite.com 2000]
```

<table>
<tr><td></td><td><pre>set greeting [read $handle]
close $handle</pre></td></tr>
<tr><td colspan="2">Make a connection to a network socket (act as a client) and retrieve data. The example assumes a server is listening on TCP port 2000 of the specified host.</td></tr>
<tr><td><pre>(No equivalent)</pre></td><td><pre>proc greeting {handle client_ip client_port} {
        puts $handle {Welcome to our greeting server!}
        close $handle
}
socket −server greeting 2000</pre></td></tr>
<tr><td colspan="2">Implement a network server to answer the client shown above. This is the complete script. If you're using Wish (the Tcl windowing shell) this will run all day as shown. If you're using Tclsh (the console Tcl shell) add a vwait command at the end, to make the program wait for events instead of terminating at the end of the script. That difference between the two shells is necessary and intentional, since Wish is event−driven by default, and Tclsh is not.</td></tr>
</table>

3. Getting More Information

- *General Tcl/Tk programming and introduction:* See Brent Welch's unbelievable book *Practical Programming in Tcl and Tk*. Due to Brent's generosity, you can even read and print the older editions and selected chapters from the current editions at http://www.beedub.com/book .
- *Downloads needed to develop in Tcl:* See http://www.tcl.tk for TclPro 1.4.1 for all operating systems, plus almost any add−on package you could ever want. TclPro contains the 2 interpreters (Tclsh and Wish) version 8.3, plus an excellent interactive debugger and a suite of helpful tools and libraries. Version 1.4.1 was released to the public. However, as of mid−2002, it looks like ActiveState is taking over the TclPro product as a commercial product. Remember you can always get the 'standard' interpreters for all operating systems from http://tcl.sourceforge.net because Tcl is open source software.
- *Editors with syntax highlighting, etc:* For MS Windows, I like the inexpensive commercial product TextPad at http://www.textpad.com. Currently the cost is $27 US per license, and you can try before you buy. Be sure to get the Tcl syntax definition file from their web site. TextPad is the most feature−rich editor for MS Windows I've ever seen, and has the ability to emulate Microsoft editors' behavior. You can use it as an IDE for Tcl/Tk development by interfacing it with the interpreters and your other tools. For Unix/Linux, and maybe even for MS Windows, try Nedit at http://www.nedit.org. It's free under the GNU General Public License. It also does a good job of making MS Windows users productive right away.
- *Tools you'll probably want:* The first thing most VB programmers want is to hit an ODBC database. Go get the TclODBC 2.2 package from http://www.tcl.tk . It's a DLL for Win32 that hooks you into all ODBC data sources and drivers. It comes with documentation, and there's a minimal example above. Note that it may or may not be portable to other operating systems, so you might want to wrap all your calls to it into procedures. That way you can port your code to use other libraries later. Regular expressions are almost a powerful programming language of their own. Accordingly, they take some time to master. The simple Tcl program 'Visual RegExp' has helped me tremendously with that. Get it at http://laurent.riesterer.free.fr/regexp . There are also several packages available for hooking Tcl to the world of ActiveX, so you can automate MS Office applications, etc..
- *Essential help topics:* Once you have TclPro and its help file, go to its index and visit the 'Tcl' topic. There's a concise summary of the language's syntax rules, and the substitutions that drive it. Also be sure to hit the 're_syntax', 'tclvars', 'tclsh', and 'wish' topics. These are apparently translated from the Tcl man pages on Unix/Linux, and are some of the best texts I've ever seen for WinHelp, if you need *reference material*. I don't recommend reading this help file as your first introduction, but it is an excellent reference while programming.
- *'Start' menu items:* Once you have TclPro installed, you should look at the 'Start' menu for TclPro, and check out the 'Incr Widgets Reference' and 'Widget Tour'. These show the built−in GUI capabilities of Tk *with the actual Tcl code required to use them.*
- *Advocacy (how to convince your management to use Tcl/Tk):* A wealth of advocacy information is available at http://www.tcl.tk .

---

4. Copyright and License