Scripting Graphical Commands with Tcl/Tk Mini-HOWTO

Salvador J. Peralta <sperالta [at] willamette [dot] edu>

Revision History
Revision 1.0 release 2003-08-19 sjp
Subtitle: Putting a graphical face on commandline utilities with Tcl and Tk.

Abstract

One of the richest aspects of Linux is its abundance of commandline utilities. The ability to rapidly provide a graphical frontend for those utilities to make them available to non-technical users can be a handy skill for a developer or administrator to possess. This article provides a cookbook-style tutorial introduction to Tcl and Tk, a scripting language and graphical toolkit that were designed to accomplish that very task.

Table of Contents

Copyright

Introduction to Tcl and Tk

The tool command language Tcl (pronounced tickle) is an interpreted, action-oriented, string-based, command language. It was created by John Ousterhaut in the late 1980's along with the Tk graphical toolkit. Tcl and the Tk toolkit comprise one of the earliest scripted programming environments for the X Window System. Though it is venerable by today's standards, Tcl/Tk remains a handy tool for developers and administrators who want to rapidly build graphical frontends for command line utilities.

Tcl and Tk come bundled with most major Linux distributions and source-based releases are available from tcl.sourceforge.net. If Tcl and Tk are not installed on your system, the source releases are available from the SourceForge Tcl project: http://tcl.sourceforge.net/. Binary builds for most Linux distributions are available from rpmfind.net. A binary release is also available for Linux and other platforms from Active State at http://aspn.activestate.com/ASPN/Tcl.

Tcl and Tk Basics

Tcl is built up from commands which act on data, and which accept a number of options which specify how each command is executed. Each command consists of the name of the command followed by one or more words separated by whitespace. Because Tcl is interpreted, it can be run interactively through its shell command, **tclsh**, or non-interactively as a script. When Tcl is run interactively, the system responds to each command that is entered as illustrated in the following example. You can experiment with tclsh by simply opening a terminal and entering the command **tclsh**.

```
$ tclsh
% set a 35
35
% expr 35 * $a
1225
% puts "35 * a is: [ expr 35 * $a ]"
35 * a is: 1225
% exit
$
```

The previous example illustrates several aspects of the Tcl language. The first line, **set a 35** assigns 35 to the variable **a** using the **set** command. The second line evaluates the result of 35 times the value of **a** using the **expr** command. Note that Tcl, like Perl and Bash requires the use of the dollar operator to get the value of a variable. The open brackets around the expression **[expr 35 * $a]** instruct the interpreter to perform a command substitution on the expression, adds it to the rest of the string and uses the puts command to print the string to Tcl's default output channel, standard output.

Tcl's windowing shell, **Wish**, is an interpreter that reads commands from standard input or from file, and interprets them using the Tcl language, and builds graphical components from the Tk toolkit. Like the **tclsh**, it can be run interactively.

To invoke Wish interactively, start X on your system, open a terminal, and type **wish** at the command prompt. If your environment is set up properly, this will launch an empty root window and start the windowing shell in your terminal. The following example is a two-line script that is one of the simplest programs that can be created with **wish**:

```
$ wish
% button .submit -text "Click Me" -command { puts "\nHello World" }
.submit
% pack .submit
```

Let's break down these two lines of code:

button .submit -text "Click Me" -command { puts "\nHello World" }:

The **button** command enables you to create and manipulate the Tk button widget. As with all Tk widgets, the syntax is **button .name [-option value] [-option value] ...**. The curly braces surrounding the **puts** command allow you to nest the text string, "Hello World", inside of the command without performing any variable substitutions. Other basic widgets include the following: label, checkbutton, radiobutton, command, separator, entry, and frame. Click the button a few times to verify that it works.

pack .submit

The **pack** command tells the Tk packer geometry manager to pack the window name as a slave of the master window **.** which is always referred to by the character **.**. As with the other Tk widget commands we will see, the syntax is **pack .name [-option value] [-option value]**.

While the previous example was very simple, more advanced examples are nearly as easy to build. Have a look at the following script which creates a simple graphical front end for apachectl (please note, this example is intended to be run as a script rather than interactively from the shell. You will need to set the permissions of the script as executable and run this script as a user with privileges to start and stop apache):

```
#!/usr/bin/wish

set apachectl "/usr/local/apache/bin/apachectl"
global apachectl

proc start {} {
  global apachectl
  exec $apachectl start &
}

proc stop {} {
  global apachectl
  exec $apachectl stop &
}

proc restart {} {
  global apachectl
  exec $apachectl restart &
}

proc screen {} {
  frame .top -borderwidth 10
  pack .top -fill x
  button .top.start -text "Start" -command start
  button t.op.stop -text "Stop" -command stop
  button .top.restart -text "Restart" -command restart
  pack .top.start .top.stop .top.restart -side left -padx 0p -pady 0
}
screen
```

This script introduces a few new concepts. Let's look at some of them line by line:

```
set apachectl "/usr/local/apache/bin/apachectl"
global apachectl
```

As we saw earlier, the **set** command is used to assign a value to a variable. As with the previous examples, the syntax is simple: **set variable_name value**. In order to make the variable available to the Tcl procedures that we are creating in this program, we need to import the apachectl variable into each procedure. This is accomplished using the **global** command which adds a named variable to the local namespace of a given procedure. The **global** command accepts one or more variables as arguments and assigns the named variables to each procedure used in the program. Global is also used to export variables that are declared within a procedure's local namespace.

```
proc start {} {
  global apachectl
   exec $apachectl start &
}
```

Procedures in Tcl are created with the **proc** command. The **proc** command takes the following form: **proc name {args} {body}** where name is the name of the procedure. Args are the formal arguments accepted by the procedure, and body is the main code of the procedure. Procedures are executed the same way that any other command is executed in Tcl.

The script we are currently working with consists of 4 procedures. The first 3 (start, stop, restart), simply import the apachectl variable into the local namespace and execute the basic apachectl commands as background processes while the 4th procedure, "**screen**", uses the packer to build the basic screen and call each of the functions.

Let's have a closer look at the **screen** procedure:

```
proc screen {} {
  frame .top -borderwidth 10
  pack .top -fill x
  button .top.start -text "Start" -command start
  button .top.stop -text "Stop" -command stop
  button .top.restart -text "Restart" -command restart
  pack .top.start .top.stop .top.restart -side left -padx 0p -pady 0
}
```

The **screen** procedure begins by using the **frame** command to construct the basic frame that will contain the buttons specified further down in the procedure. As this example illustrates, slave widgets are specified by prepending them with the name of their master followed by a ".". The master must already be packed before the slave can use them, so we pack the frame .top before specifying the **button** command and tell it to fill along the x axis.

Last, we use the **button** command to create 3 buttons as slaves to **.top**, passing in the appropriate procedure to execute when the button is pressed, and adding a text label using the **-command** and **-text** arguments, respectively.

Adding Features

Providing multiple buttons to control a single application is, perhaps, a bit of overkill, as is calling separate procedures for each action. A third problem is that apachectl prints a message to standard output to indicate how the command has been acted upon. The application could be improved by including a text widget to display the output of apachectl.

In the following script, we will redesign the application to use a radiobutton chooser and a single button by modifying the **screen** procedure , and build a text widget in a new frame. We also remove the start, stop, and restart procedures and create 2 new procedures. The first, **init**, will handle the conditionals created by the radio button selection, the second, **put_text**, will launch Apache and print the apachectl output to a text widget:

```
#!/usr/bin/wish

set apachectl "/usr/local/apache_new/bin/apachectl"
```

```
proc screen {} {
  frame .top -borderwidth 10
  pack .top -fill x
  radiobutton .top.start -text "start" -variable mode -value start
  radiobutton .top.stop  -text "stop" -variable mode -value stop
  radiobutton .top.restart -text "restart" -variable mode -value restart
  button .top.submit -text execute -command init
  pack .top.start .top.stop .top.restart .top.submit -side left -padx 0p -pady 0 -
  frame .bottom
  pack .bottom -fill x
  text .bottom.main -relief sunken -bd 2 -yscrollcommand ".bottom.scroll set"
  scrollbar .bottom.scroll -command ".bottom.main yview"
  pack .bottom.main -side left -fill y
  pack .bottom.scroll -side right -fill y
}

proc init { } {
 global mode action
 switch $mode {
    stop     {set action "stop"}
    restart  {set action "restart"}
    default  {set action "start"}
  }
 put_text
}

proc put_text {} {
  global action apachectl
  set f [ open "| $apachectl $action" r]
  while {[gets $f x] >= 0} {
    .bottom.main insert 1.0 "$x\n"
  }
  catch {close $f}
}
screen
```

First, let's have a look at the **screen** procedure. The **radiobutton** command works just like html radiobuttons. The **-variable** parameter accepts the name of the variable as an argument. The **-value** parameter accepts the variable's value as an argument. The button, .top.submit uses the **-command** parameter to call the init procedure defined later in the script. These buttons are then packed into the top frame and a second frame called bottom is created.

The bottom frame is composed of a text widget and a scrollbar. Text widgets are created with the **text** command which takes a variety of options. In this case, we have used the **-relief** option which specifies the 3D effect for the field (other values for -relief include raised, flat, ridge, solid, groove); **-bd** option, which specifies borderwidth; and the **yscrollcommand** which specifies the name of a scrollbar that will be engaged by the textfield. Our scrollbar widget takes one option, **-command** which specifies how to behave when text scrolls beyond the screen of the text widget that it is interacting with.

The **init** procedure loads the mode variable into its local namespace using the **global** command and uses a **switch** statement to set the value of the global variable, **action**.

In this example, the **switch** command tests whether "$mode" matches the first word on each line in the list, and performs the action specified on the second word of each line. The default value is specified at the bottom of the list and defines the action performed if no match is found. Switch accepts 4 options: **-exact**, which requires a case-sensitive match, **-glob**, which uses a glob-style pattern match, **-regexp**, which uses regular-expression style matching, and **--**, which indicates the end of options, and is typically used if the pattern being matched has a "-" as a prefix.

Note: We could have used an if-elseif-else conditional chain rather than the switch statement:

```
if { $mode == "stop" } {
  set action "stop"
} elseif { $mode == "restart" } {
  set action "restart"
} else {
  set action "start"
}
```

The final thing that the **init** procedure does is call the **put_text** procedure.

The **put_text** procedure reads in the value of action that was set in the init procedure, executes apachectl with the appropriate argument as specified by action, and prints apache's output to the .bottom.main text widget.

```
proc put_text {} {
  global action apachectl
  set f [ open "| $apachectl $action" r]
 while {[gets $f x] >= 0} {
   .bottom.main insert 1.0 "$x\n"
 }
}
```

The **put_text** procedure introduces 3 new commands:

First, it sets the value of a variable, f, to the output of the open command. **Open** can be used to open a file, pipe stream or serial port and returns an identifier which can be used for reading, writing, or closing a stream. Since the first character following the **open** is a pipe "|", **$apachectl $action** is treated as a command, and is executed as though the **exec** had been given. The **r** specifies that the stream is read-only. Other parameters are as follows:

```
r read only
r+  read and write if file exists
w write only
w+  read and write if file exists
a write only.  Create new file if none exists.
a+  read and write. Create new file if none exists.
```

The second new command is **while**. While is a typical while loop which executes a body of arguments so long as the specified condition is met. In this case, **while** will read a line of input and save it to the variable **x** until there is nothing left to read. The insert command inserts each line of input to the zero'th character of line 1 (1.0) of the .bottom.main text widget.

Conclusions

This was a fairly quick and dirty introduction by example to creating graphical user interfaces for command line applications with Tcl and Tk. In addition to the language's basic syntax, we learned about procedures, several of the basic widget sets, progam control flow, some of Tcl's logical operators, and the basics of how to insert text into a text widget.

The simplicty of Tcl's syntax makes it a very easy language to learn and to build in. Our final example was built with less than 40 lines of code. If you want to go beyond the basics described here, the gridded window provides a more advanced mechanism for building window geometries than the packer that we used for this tutorial. Also, Tcl can be used for more than simply scripting graphical interfaces. For example, Don Libes' expect programming language, which provides facilities for constructing a programmed dialogue with interactive programs, was written in Tcl and C. The next section discusses where to go next for more resources on Tcl and Tk.

Further Reading

Readers interested in learning more about Tcl and Tk are encouraged to followup this article with some of the resources listed below:

1. Tcl comes with an excellent online manual system. Tcl and the Tcl shell, tclsh, are described with "man Tcl" and "man tclsh" respectively. All commands that are part of the Tcl base environment are described in the "n" section of the manual. Readers interested in learning more about optional parameters to Tcl commands are encouraged to make use of the "options" man page.

2. John Ousterhaut's "Tcl and the Tk Toolkit" is in its 14th printing, and remains a classic programming text. In addition to providing the basics of Tcl and Tk scripting, "Tcl and the Tk Toolkit" also provides a great deal of information about how to extend Tcl and Tk in the C programming language.

3. Brent Welch's "Practical Programming in Tcl and Tk" is now in its fourth edition and is a very supplementary good reference to the Ousterhaut book.

4. comp.os.lang.tcl is the primary newsgroup for people using Tcl and Tk.

5. Scriptics.com is the primary online resource for the Tcl/Tk developer community.

6. Cameron Laird maintains an excellent online resource of Tcl and Tk tutorials at http://phaseit.net/claird/comp.lang.tcl/tcl_tutorials.html. [http://phaseit.net/claird/comp.lang.tcl/tcl_tutorials.html]

About the Author

Salvador Peralta is the Systems Administrator for the Mark O. Hatfield Library at Willamette University. He is a regular contributor to the Linux Documentation project and lives in Oregon with his wife and two dogs. Salvador welcomes comments and questions via email to speralta [at] willamette [dot] edu.