Tcl/Tk quick start

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Who should take this tutorial?

This tutorial is designed for those with experience in one or more programming or scripting languages. While Tcl/Tk is available on several platforms, including Win32 and MacOS as well as several of the *NIX environments, this tutorial is written in the context of running on a GNU/Linux installation.

Starting out, I'll introduce Tcl/Tk and summarize a small part of the language's history. Then, I'll review key features of the Tcl/Tk scripting language and interpreter, discuss some of the extensions to the language, and examine several examples of Tcl/Tk in action. The text is accompanied by code fragments and occasionally an image of the resulting output (since Tk is a GUI toolkit, after all).

Finally, I'll wrap up with a presentation of some external resources, both Web and print, to help you deeper into the Tcl/Tk environment.

About the author

Brian Bilbrey is a system administrator, product engineer, webmaster, and author. Linux is a tool in his daily work as well as his avocation, much to the chagrin of his long-suffering spouse, Marcia.

He welcomes your feedback on this tutorial or other related Linux topics at *bilbrey@orbdesigns.com*. His daily journal on life with Linux and other adventures can be found at *OrbDesigns.com*.

Section 2. Tcl/Tk basics

Origins of Tcl/Tk

Tcl stands for Tool Control Language. Tk is the Graphical Toolkit extension of Tcl, providing a variety of standard GUI interface items to facilitate rapid, high-level application development.

Development on Tcl/Tk, pronounced "tickle tee-kay", began in 1988 by John K. Ousterhout (shown in the image), then a Professor at U.C. Berkeley (UCB). Tcl was designed with the specific goals of extensibility, a shallow learning curve, and ease of embedding. Tk development began in 1989, and the first version was available in 1991. Dr. Ousterhout continued development of Tcl/Tk after he left UCB, then going to work for Sun Microsystems for a stint. Now at *Scriptics* (which begat Ajuba Solutions, which was purchased by Interwoven), he keeps on improving the language, currently in version 8.3.2 stable and 8.4 development, as of this writing.

See the "*History of Tcl*" page for more details.

Tools and files

There are two main programs that you need on your Linux system to explore Tcl/Tk. These are *tclsh* and *wish*. As you might discern from its name, the former is a Tcl shell, most frequently used to provide execution context for a shell script. Wish is the equivalent, for a windowed GUI environment.

Check for the presence of these files by typing the following:

```
~/tcltk$ which tclsh
/usr/bin/tclsh
~/tcltk$ which wish
/usr/bin/wish
```

The which command returns the path to the specified executable. If you don't see results similar to these, then you'll want to head over to the *Scriptics Tcl/Tk page* to download and build your own copy. Of course, the absence of these programs on your system is not indicative of any problem. Unlike Perl, Tcl/Tk is generally not regarded as essential to the operation of Linux. Every distribution I'm aware of ships with a version of Tcl/Tk and the most popular extensions as a part of the CDROM or online repository collection. From these sources, the tools are generally fairly easy to install. If you have difficulty, contact the publisher of your GNU/Linux software.

Section 3. The Tcl language

What makes Tcl tick?

In the following listing, you'll find a common first example program, as implemented in Tcl. This is a complete script: the first line invokes the tclsh environment, and the second does the actual work. Create the script with a text editor of your choosing, make it executable by typing **`chmod +x hello.tcl`**, then execute it to test your handiwork.

```
~/tcltk$ cat hello.tcl
#!/usr/bin/tclsh
puts stdout {Hello, World!}
~/tcltk$ ./hello.tcl
Hello, World!
```

Tcl and Tk are interpreted, extensible scripting languages. The license, which is very similar to the BSD license, permits free use under any circumstances, as long as the copyright is retained in all copies and notices pass verbatim in any distribution. The license terms make Tcl/Tk free software.

Tcl/Tk is an interpreted environment. The Tcl interpreter can be extended by adding pre-compiled C functions, which can be called from within the Tcl environment. These extensions can be custom for a specific purpose, or generic and widely useful. We'll look at a number of extensions later in the tutorial, with special attention given to the first extension, the very popular *Expect*.

In next few panels, we'll review the major features of the Tcl language, from metacharacters and global variables, to operators, mathematical functions, and core commands. After all, the

commands make Tcl/Tk the distinctive evolving language it is. Bear in mind that we do not have the space in this tutorial to cover every command. The highlights are here, the depths are yours to explore at a later date.

```
#!/usr/bin/tclsh
# filename hello2.tcl
# This program code shows
# metacharacter usage
puts stdout "Hello, World! \a"
puts stdout {Hello, World! \a}
set Pints 6
set Days 7
puts stdout "The answer to the \
 universe is [eval $Pints * $Days] \n"
***
~/tcltk$ ./hello2.tcl
Hello, World!
Hello, World! \a
The answer to everything is 42!
```

Tcl metacharacters

Metacharacters are those characters or character pairs that have special meaning in the context of the Tcl/Tk environment, including grouping statements, encapsulating strings, terminating statements and more, as delineated in the following table. Many of these are demonstrated in the code listing to the left. One special feature to notice is the difference in output when curly braces (which prevent substitution and expansion) are used in place of double quotes.

Character(s)	Used as
#	Comment
; or ***newline***	Statement separators
Name	A variable (case sensitive)
Name**(*****idx*****)**	Array Variable
Name**(*****j,k,l...*****)**	Multidimensional Array
"*string***"**	Quoting with substitution
{*string***}**	Quoting without substitution
[*string***]**	Command substitution
char	Backslash substitution
\	Line continuation (at end of line)

```
#!/usr/bin/tclsh
#
# Demonstrate global variables
# and backslash substitution
if {$argc >= 1} {
    set N 1
    foreach Arg $argv {
        puts stdout "$N: $Arg\n"
        set N [expr $N + 1]
        if {$Arg == "ring"} {
            puts stdout "\a"
        }
    }
} else {
    puts stdout "$argv0 on \
        X Display $env(DISPLAY)\n"
}
***
~/tcltk$ ./hello3.tcl
```

Tcl global variables and backslash substitutions

Several global variables exist (and are pre-defined, if not null in the current context) when a Tcl/Tk script begins running. These variables permit access to the operating environment as follows: *argc* is the count of arguments to the script, not counting the name as invoked. *argv* is a list (not an array) of arguments. *argv0* is the invoked filename (which may be a symlink). *env* is an array that is indexed by the names of the current shell's environment variables. *errorCode* stores information about the most recent Tcl error, while *errorInfo* contains a stack trace from the same

```
./hello3.tcl on  X Display :0.0
~/tcltk$ ./hello3.tcl ring
1: ring
```

event. The list goes on with another dozen *tcl_xxx* variables from `tcl_interactive` to `tcl_version`. A good summary is found in *Tcl/Tk in a Nutshell* (see the Resources at the end of this tutorial for more information).

Several of these variables are used in the sample code at left, along with (once again) some backslash-quoted characters, \n and \a. *char* allows substitution of non-printing ASCII characters. This is common to many scripting and shell environments under UNIX. As noted in the table, a backslash-quoted character that has no defined substitution is simply echoed to output.

\character	Substitution
\a	Bell
\b	Backspace
\f	Formfeed
\n or \newline	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical Tab
\space ("\ ")	Space
ddd	Octal value
\x*ddd*...	Hexadecimal value
\c	Echo 'c'
****	Backslash

```
~/tcltk$ cat maths.tcl
#!/usr/bin/tclsh
#
# Demonstrate operators and
# math functions
set PI [expr 2 * asin(1.0)]
if {$argc == 3} {
  set X [lindex $argv 0]
  set Y [lindex $argv 1]
  set Rad [lindex $argv 2]
  set Dist [expr sqrt(($X*$X)+($Y*$Y))]
  set Cir [expr 2*$PI*$Rad]
  set Area [expr $PI*$Rad*$Rad]
  puts stdout "Distance = $Dist"
  puts stdout "Circumference = $Cir"
  puts stdout "Area = $Area"
} else {
  puts stdout "Wrong argument count!"
  puts stdout "Needs X, Y, and Rad(s)"
}
********
~/tcltk$ ./maths.tcl 3 4 5
Distance = 5.0
Circumference = 31.4159265359
```

Tcl operators and mathematical functions

Tcl supports a standard array of operators and mathematical functions. The operators include arithmetic, bitwise, and logical operators, which are evaluated via the *expr* command using common operator precedence rules. Also, considering Tcl's fundamental roots as a string-oriented scripting language, there's a reasonable complement of mathematical functions as follows:

* Trigonometric functions include cos(x), acos(x), cosh(x), sin(x), asin(x), sinh(x), tan(x), atan(x), atan2(y, x), tanh(x), and hypot(x, y). The pertinent unit for these functions is radians.
* The Log functions are exp(x), log(x), and log10(x).

```
Area = 78.5398163397
```

* The Arithmetic functions are ceil(x), floor(x), fmod(x, y), pow(x, y), abs(x), int(x), double(x), and round(x).
* Random numbers are handled by rand() and srand(x).

The example at left makes use of just a few of these operators and functions to calculate the distance between a specified point and the origin, and to return the circumference and area of a circle of the specified radius. Additionally, in this example we use the list index (*lindex*) command to access individual elements of $argv.

Looping and branching in Tcl

The looping commands in Tcl are *while*, *for*, and *foreach*. The conditional (branching) commands are *if/then/else/elsif* and *switch*. Modifiers for the preceding commands are *break*, *continue*, *return*, and *error*. Finally, the *catch* command is provided for error handling.

if/then/else/elsif was demonstrated in previous panels. While *then* is a part of the formal syntax, it is most often observed in absentia.

In the example at the left, a *switch* command is fed the command line arguments by the *foreach*

```
...
#
# parse command line switches
set Optimize 0
set Verbose 0
foreach Arg $argv {
        switch -glob -- $Arg {
                -o* {set Optimize 1}
                -v* {set Verbose 1}
                default {
                        error "Unknown $Arg"
                }
        }
}
set LineCount 0
while {[gets stdin Line] >= 0} {
        # to confuse Vanna White...
        Remove_Vowels $Line \
                $Optimize $Verbose
        incr LineCount
}
return LineCount
...
```

construct. As the arguments are processed (Note: improper input terminates the script, since we haven't implemented an error *catch*), the *while* loop processes input by calling a procedure for each line, and *incr*ementing a line counter. The code fragment ends by *return*ing the count of lines processed.

```
~/tcltk$ tclsh
% set Phrase "hello, world!"
hello, world!
% string toupper $Phrase
HELLO, WORLD!
% string totitle $Phrase
Hello, world!
% string match ello $Phrase
0
% string match *ello* $Phrase
1
% string length $Phrase
14
% append Phrase "Nice day, eh?"
hello, world!
Nice day, eh?
% string toupper $Phrase
HELLO, WORLD!
NICE DAY, EH?
% string wordend $Phrase 7
12
```

Tcl strings and pattern matching

Strings are the fundamental data type in Tcl. The *string* command is really a variety of commands, gathered under one umbrella. In use, *string* reads much like the application of specific object methods from OOP programming, as you can see in the example on the left.

The informational *string* commands are *length* and *bytelength* (which can differ, depending on character set). Comparisons that return boolean values (1 or 0) are *compare*, *equal*, and *match*. Pattern matching here is accomplished by "globbing", the simple type of matching commonly associated with shell operations. Advanced Regular Expressions are also available via the distinct regex and regsub commands.

Indexing functions in Tcl are performed with the *index*, *last*, *first*, *wordend*, and *wordstart* commands. String modification is handled by

tolower, *toupper*, *totitle*, *trim*, *trimleft*, *trimright*, *replace*, and *map*. The latter requires a character mapping table to be pre-defined. Substrings are extracted with *range*, and strings are output multiple times with *repeat*.

Text can be added to an existing variable using the *append* command. The *format* command can be used to generate output strings using the same styles and conventions as the C language's printf command. *scan* parses a string and assigns values to variables. Finally, starting with Tcl 8.0, functionality for handling binary data as strings (thus able to process the null character without failure) was added, with the *binary format* and *binary scan* commands.

```
~/tcltk$ tclsh
% set c1 {Bob Carol}
Bob Carol
% set c2 [list Ted Alice]
Ted Alice
% set Party1 [list $c1 $c2]
{Bob Carol} {Ted Alice}
% set Party2 [concat $c1 $c2]
Bob Carol Ted Alice
% linsert $Party1 1 Richard
{Bob Carol} Richard {Ted Alice}
%
```

Tcl lists

Lists have two major uses in Tcl. The first we've already seen demonstrated in the context of processing command line arguments via the *foreach* command (found on Looping and branching in Tcl on page 7). The second use is to build up elements of a Tcl command dynamically, which can be later executed by using the eval command, as we see later in this tutorial.

The *list* command takes all of its arguments and returns them in a list context. Arguments may be values or variables. From the example at left, lists can be created manually, or using list, which can take other lists as arguments (thus saving the two couples orientation of our first "Party"). Alternatively, the *concat* command is used to merge two or more lists into a single entity of top-level items, returning the second, more interesting "Party".

Some other useful list commands and their syntax are:

* **llength *$List*** - returns count of toplevel items.
* **lindex *$List n*** - returns indexed item, counts from zero.
* **lrange *$List i j*** - returns range of list elements.
* **lappend *$List item*...** - append items to list.
* **linsert *$List n item*...** - inserts item(s) at the specified position in the list, moving other items down the list.

The balance of list commands include *lreplace*, *lsearch*, and *lsort*. The *split* command takes a string as input and generates a properly parsed list, breaking the string at the specified character. *join* performs the complementary operation, taking list elements and stringing them together, separated by a *joinstring*.

Tcl arrays

The shortcut to understanding Tcl arrays is to regard them in the same light as you would a Perl hash. The array is not a numerically indexed linear data structure, unless you choose to impose that interpretation upon your data. The index (or key) may be any string, although strings with spaces need either to be quoted, or a variable reference.

Just as with normal variables, arrays are initialized with the *set* command, as shown at left. The index

```
~/tcltk$ tclsh
%  set People(friend) Tom
Tom
%  set People(spouse) Marcia
Marcia
% set People(boss) Jack
Jack
% array names People
friend boss spouse
% set Person $People(friend)
Tom
% array get People
friend Tom boss Jack spouse Marcia
% set People(friend) \
    [concat $People(friend) Bob]
Tom Bob
% set Person $People(friend)
Tom Bob
%
```

is given inside parentheses. Please note that the parentheses do not provide grouping like curly brackets or double quotes. Once initialized as an array, a variable may not be accessed as a singular variable. As shown at the bottom of the listing to the left, array elements may also be lists.

More Tcl arrays

The *array* command is a multi-purpose tool, much like *string*. The commands are *array exists* to test a variable for existence as an array, *array get* to convert to list format, *array set* to convert from list to array, *array names* to return a list of indices, and *array size* to return the count of indices. Searching through an array has its own set of four commands: *array startseach*, *array anymore*, *array nextelement*, and *array donesearch*.

Although Tcl arrays are one-dimensional by design, there is an elegant way to simulate multi-dimensional constructs. Since indices are arbitrary strings, a 2D array might be declared as follows:

```
set i 1 ; set j 10
set array($i,$j) 3.14159
incr $j
set array($i,$j) 2.71828
```

These array keys are really just the strings "1,10" and "1,11" respectively, but for the purposes of accessing the data, who's to know the difference?

Tcl procedures

The *proc* command defines a Tcl procedure. Once

```
#!/usr/bin/tclsh
#
# Demonstrate procedures and
# global scoping briefly
set PI [expr 2 * asin(1.0)]
proc circum {rad} {
        global PI
        return [expr 2.0 * $rad * $PI]
}
proc c_area {rad} {
        global PI
        return [expr $rad * $rad * $PI]
}
set rad 3
puts stdout "Area of circle of\
radius $rad is [c_area $rad],\n\
the circumference is\
[circum $rad].\n"
*********
~/tcltk$ ./protest.tcl
Area of circle of radius 3 is 28.2743338823,
 the circumference is 18.8495559215.
```

defined, a procedure can be called or used just as a built-in Tcl command would be. Additionally, parameters can be defined with default values; for example, changing the definition at the left to read `proc c_area { {rad 1} }` would enable the c_area procedure to be called without parameters, returning the area of the unit circle.

The *rename* command is used just as it sounds, to give a new name to an existing command or procedure. There are two distinct reasons for using rename. The first is to add functionality to an existing command by renaming the original, then replacing it with a procedure of the same name. The procedure can call the original, and add whatever bells and whistles are necessary. The second reason for rename is to map a command out of existence, like `rename exec {};`, for example, to prevent users from executing external commands.

Variable scoping rules

Rules of scope describe the visibility of procedure and variable names and values at different levels of the program. For instance, variables defined at the outermost level of a script are global variables. By default, these are not visible, nor are their values available from within procedures. This permits procedure writers to freely define variable names and assign values without fear of overwriting important variables that are unknown at the local scope. To make a global variable visible inside a procedure, it must be declared as such *within the procedure*, as I did for PI (in the example on the previous panel) using the *global* command.

The *upvar* command provides a facility for associating a local-level variable with the value of a variable from another scope. This permits calling by name into procedures, which is handy when the procedure needs to be able to modify the value at another scope, rather than just using it. The command syntax is **upvar *level $VarName LocalVar***, where level is the number of steps up out of the current scope. "#0" is the representation of the global scope level.

Data structures in Tcl

Beyond simple multi-dimensional arrays, it is generally recommended that Tcl data structures be implemented as arrays that have dedicated procedural interfaces. This design hides specific implementation details from the user of the structures, while providing the ability to perform significant error checking capabilities.

In the example at left, after declaring *uData* as a

```
#!/usr/bin/tclsh
#
# Demonstrate Data Structures
# using procedural wrappers
proc UserAdd { Acct rName eMail phone } {
  global uData
  if {[info exists uData($Acct,rname)]} {
    return 1
    }
  set uData($Acct,rname) $rName
  set uData($Acct,email) $eMail
  set uData($Acct,phone) $phone
  return 0
}
```

```
puts stdout [UserAdd bpb\
  Brian bilbrey@junk.com 555-1213]
puts stdout [UserAdd tom\
  Tom tom@junk.com 555-1212]
puts stdout [UserAdd bpb\
  Brian bilbrey@junk.com 555-1213]
******
~/tcltk$ ./datas.tcl
0
0
1
```

global variable, the code executes a check to see that the account doesn't already exist. If it does, then the procedure returns with a (non-zero) error message. The return can be used in a switch to generate error text output. For the example, we simply feed it three sequential inputs, including one repeat. This yields the output shown at the bottom, with the '1' indicating a purposeful error return due to a repeating account name.

Other possibilities for data structures include lists of arrays, linked or doubly-linked arrays, or various permutations thereof. The lists of arrays construct are considerably more efficient with the list reimplementation that accompanied Tcl 8.0, providing constant access times.

```
~/tcltk$ tclsh
% file exists hello3.tcl
1
% file executable testit
0
% file pathtype ./hello3.tcl
relative
% set dir1 home
home
% set dir2 brian
brian
% set dir3 tcltk
tcltk
% file join /$dir1 dir2 dir3
/home/dir2/dir3
% file delete testit~
%
```

Paths and files

File and path operations are a challenging problem in a cross-platform environment. Tcl uses UNIX pathnames (separated using the '/' character by default), as well as the native pathname construction for the host OS. Even when in-program data is constructed properly, it can be difficult to ensure that user input matches the system requirements. The *file join* command is used to convert UNIX formats to native pathnames. Other path string commands include *file split*, *dirname*, *file extension*, *nativename*, *pathtype*, and *tail*.

In its role as a "Tool Control Language", Tcl has a wide variety of internal file test and operation features. Each command leads off with *file*, as in

file exists name. Other test commands (which all return boolean values) include *executable*, *isdirectory*, *isfile*, *owned*, *readable*, and *writable*.

File information and operations are accomplished (again, all with the leading *file*) *atime*, *attributes*, *copy*, *delete*, *lstat*, *mkdir*, *mtime*, *readlink*, *rename*, *rootname*, *size*, *stat*, and *type*. Note that a number of the file information commands may return undefined data when running in the Windows or Mac environments, since, for example, inode and symbolic (*and* hard) link data isn't represented in those file systems.

The advantage of using *file ...* commands rather than using native commands via *exec* is that the former presents a portable interface.

This whole document could easily be devoted to just this one section of the Tcl language. However, content yourself with the *tclsh* examples to the left, which reveal the flavor of these commands, and then follow up with readings from the Resources listed at the end of the tutorial.

Processes and file I/O with Tcl

The *exec* command is used to explicitly execute

```
~/tcltk$ tclsh
% nslookup orbdesigns.com
Server:         192.168.1.3
Address:        192.168.1.3#53
Name:   orbdesigns.com
Address: 64.81.69.163
% set d [date]
Sun Mar 25 13:51:59 PST 2001
% puts stdout $d
% set d [exec date]
Sun Mar 25 13:52:19 PST 2001
% puts stdout $d
Sun Mar 25 13:52:19 PST 2001
******
% if [catch {open foo r} Chan] {
    puts stdout "Sorry, Dave...\n"
}
% gets $Chan
One
% gets $Chan
Two
% eof $Chan
0
% close $Chan
%
```

external commands. Under Linux, most external commands can be run directly when Tcl is in interactive mode, as shown in the example at the left. Running with *exec* returns the stdout output of a program not to the screen, but to Tcl, which allows the data to be assigned to a variable. When a program is started in the background, the immediate return value is the PID for the program. *exec*'d programs can take full advantage of I/O redirection and pipelines in the UNIX environment.

Other process commands are *exit*, which terminates a running Tcl script, and *pid*, which returns the PID of current (or specified) process, handy for a variety of purposes. Tcl does not incorporate any native process control commands, but you can use the exec command in concert with PID data to accomplish many tasks.

File manipulation uses the following commands: *open*, *close*, *gets*, *puts*, *read*, *tell*, *seek*, *eof*, and *flush*. As demonstrated at left, the *catch* command is useful in error checking during file opening commands. When the program output needs to be printed before a newline character is encountered, as in a user data prompt, use *flush* to write the output buffer.

An additional feature (in supported environments) is the ability to open pipelines in the same manner that you might a file. For instance, after opening a pipeline channel with *set Channel [open "|sort foobar" r]*, the output of the first *gets* is going to be "Eight" (alphabetically, out of file data "One" through "Ten", on 10 separate lines).

```
~/tcltk$ cat input01.txt
1 + 2
4 + 5
7 - 9
~/tcltk$ tclsh
% set InFile [open input01.txt r]
file3
% while {[gets $InFile Op] >= 0} {
    set Operation "expr $Op"
    set Result [eval $Operation]
    puts stdout "$Op = $Result\n"
}
1 + 2 = 3
4 + 5 = 9
7 - 9 = -2
%
```

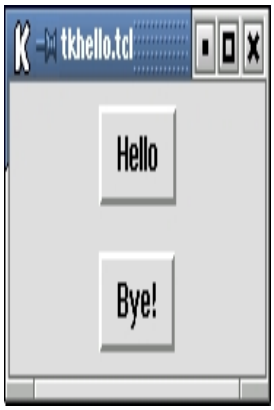
Using eval for dynamic scripting

In this example, you can sense the power of the *eval* command. Under normal circumstances, the Tcl interpreter operates in a one-pass mode: It first parses the input command line (possibly stretched over several physical lines), performing any substitutions. Then execution takes place, unless a bad or malformed command is found. *eval* permits a second pass (or perhaps more correctly, a pre-pass). Thus a Tcl command can be first dynamically constructed, then parsed and executed.

In the listing at the left, the input file consists of three lines, each with one arithmetic operation shown per line. After invoking **tclsh**, the file is opened read only, and associated with the `$InFile` variable. The `while` loop reads in one line at a time, to `$Op`. Then a complete Tcl command is constructed by pre-pending `expr` to the $Op variable. This is then expanded, evaluated, and the result assigned accordingly. Finally, each operation and result is displayed on `stdout`.

While this sample demonstrates a relatively trivial application of *eval*, conceptually it can be easily extended to dynamic file and/or directory processing based on the input of an input file of known syntax or to base operations on file type, permissions, access time or any variety of testable elements.

Section 4. Tk commands



What's a widget, anyway?

Tk is the graphical Toolkit extension for Tcl. Tk release versions are coordinated with those of Tcl. In the panels that follow, we'll review the Tk widget set, examine some of the configuration options, and set up some examples to demonstrate the useful nature of Tk.

It's difficult to convince any PHB (Pointy Haired Boss) that this section of the tutorial is work related. After all, it is about widgets, and conceptually widgets are closely related to play...but this *is* work, so let's dig into it. First, here's the code for a Tk enhanced "Hello, World!"

```
#!/usr/bin/wish
#
# Hello World, Tk-style
button .hello -text Hello \
  -command {puts stdout \
  "Hello, World!"}
button .goodbye -text Bye! \
  -command {exit}
pack .hello -padx 60 -pady 5
pack .goodbye -padx 60 -pady 5
```

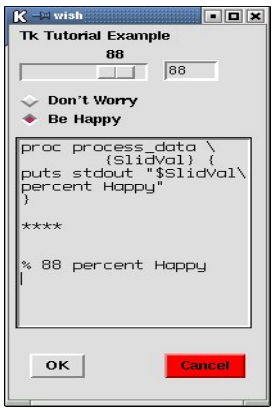
Invoking *wish* (the Tk shell) in the first line brings up a Window widget of default size. Then I defined two button widgets, *.hello* and *.goodbye* -- these are packed into the window, and the window shrinks to the size defined by the specified button spacing. When the script is executed, you get the dialog shown at the left. Click on the button to get "Hello, World!" output in the parent terminal window, Click on to terminate the script.

Tk widgets

There are remarkably few commands used in the creation of Tk widgets. Better than half are variants of button or text widgets, as you can see in the following list. Several of these items are demonstrated in the next panel.

* **button** - a simple widget with over twenty configuration options from *anchor* and *font* to *padx* and *relief*.
* **canvas** - canvas is a widget that can contain not only other widgets, but an assortment of structured graphics, including circles, lines, and polygons.
* **checkbutton** - creates a checkbox-style button widget, which is linked to a variable.
* **entry** - builds a one-line text entry box.

* **frame** - frame is a widget used primarily as either a container or spacer.
* **label** - creates a label object.
* **listbox** - creates a text string list box. Items are added after widget definition.
* **menu** - a single multi-faceted widget that can contain a variety of items in assorted menu styles.
* **menubutton** - provides a clickable toplevel interface for dropdown menu implementations.
* **message** - creates a text display window widget with features including justification and word wrap.
* **radiobutton** - creates a radio button that can be one of a set associated with a specified variable.
* **scale** - creates a slider for selecting values within a specified range and resolution.
* **scrollbar** - generates a widget (slider) for changing the portion of material (usually text or a drawing) in an associated widget.
* **text** - creates a widget that displays one or more lines of text and allows that text to be edited.
* **toplevel** - creates a new toplevel (on the X desktop) window.



A Tk demonstration

As an example of some simple Tk code, the listing below generates the image at left. The code for the procedure invoked by the OK button and sample output is shown in the text window of the image.

```
~/tcltk$ wish
%  . configure -width 200 -height 400
% label .header -text "Tk Tutorial Example"
.header
% place .header -x 5 -y 2
% scale .slider -from 1 -to 100 -orient horiz
.slider
% .slider configure -variable SlidVal
% place .slider -x 5 -y 20
% entry .slidbox -width 5 -textvariable SlidVal
.slidbox
% place .slidbox -x 120 -y 38
% radiobutton .one -text "Don't Worry" -variable Mood -va
.one
% radiobutton .two -text "Be Happy" -variable Mood -value
.two
```

```
% place .one -x 5 -y 70
% place .two -x 5 -y 90
% text .twindow -width 22 -height 14 -font {clean -14}
.twindow
% place .twindow -x 5 -y 120
% button .ok -command {process_data $SlidVal} -text "OK"
.ok
% button .cancel -command {exit} -text "Cancel" -backgrou
.cancel
% place .ok -x 15 -y 350
% place .cancel -x 120 -y 350
```

Tk commands, Part 1

There are over 20 Tk commands, that act on, enhance, or complement the Tk widget set. These include *bell*, which rings the bell, depending on the configuration of the X Window system that's running. *bind* creates an association between a Tcl script and X events; for example, a specific key-mouse combination action. *clipboard* is another of the multi-function Tk commands -- it contains all the code for clearing, loading, and pasting contents to and from the Tk clipboard (which is distinct from any clipboard features native to either X or the window manager you're using).

destroy is used to delete a window and all of its children. Used on the '.' (root) window, it deletes the entire application. *event* is a powerful tool for generating virtual window events and inserting them into the processing queue, as though the actual event (a mouse click on a button, for example) had happened for real. The *font* command is used to create specified instances of system fonts. It permits local (to the script) naming of system fonts, attribute modification of named fonts, and "deletion" of fonts. Type **font families** at a *wish* prompt for a listing of available fonts for use.

Focus is an important concept in the window management arena -- in any given display, only one window at a time may have the "attention" of the keyboard and mouse. The Tk *focus* command is used to give the script control over the display focus, sending it to specified windows. A complementary function, *grab*, allows Tk to monopolize the display focus to the point where events outside the window are reported within the window's environment. This is useful when you want to force completion of an option prior to any other system activity taking place.

Tk commands, Part 2

Continuing with our overview of Tk commands, next up is *grid*, an interface to the Tk window geometry master. It is used to order widgets in a window in a rows and columns format. *lower* (and the complementary command *raise*) mediate sub-window visibility. A lowered window obscures none of its overlapping sibling windows; a raised window is brought to the top. This is used frequently in multi-document display situations. Many of the Tk widgets and commands work from a common set of standard options. These may be viewed or added to by the *option* command.

For putting widgets and sub-windows inside a window, there are two commands, both of which have already been demonstrated: *pack* and *place*. In its simplest use, *pack* adds one or more widgets to a window, and unless otherwise instructed, shrinks the window around

those objects as we saw in the Tk Hello example at the beginning of this section. *place* sets and displays objects in a parent window, using either relative or absolute measures, for example, 5 pixels from the left side or halfway (0.5) down the window.

Other commands include *selection*, an interface to the X object selection toolset; *tk*, which provides access to selected parts of the Tk interpreter's internal state; the *winfo* command for retrieving data about Tk-managed windows; and *wm*, an interface to the running window manager, for setting a multitude of features from the title bar text to all sorts of geometry specifications and constraints.

A real (small) Tk application

I want an interface to the LAN changeover scripts that I run daily. So let's use Tcl/Tk to build a small tool for ease of use. I want it to offer selections based upon an ASCII configuration file that lives in my home directory. That file contains the data shown in the listing that follows:

```
# ~/.netsetrc
# 03.26.2001 bilbrey
# space between name and command
Home /usr/local/bin/nethome
Office /usr/local/bin/netoffice
Admin /usr/local/bin/netadmin
```

The application (a full listing and image is shown on the next panel) reads the configuration file and parses each non-blank, non-comment line for a button name and its associated action. While the script would be easier to write by defining three buttons to run explicit programs, this more general solution allows me to add just about any feature I want by only adding a single line to `~/.netsetrc`.

A drawback of the code is that it isn't tolerant of a badly formatted config file. It expects a one-word button name followed by a single space followed by the command (with arguments, if necessary) to be executed on the button press. But then, a configuration file is theoretically easier to keep in line than unstructured user input.

A sample Tk app

```
#!/usr/bin/wish
```

```
#
# netset.tcl
# 03.26.2001 bilbrey
set ConFile "~/.netsetrc"
if [catch {open $ConFile r} Conf] {
  puts stderr "Open $ConFile failed"
  return 1
}
# parse config, define buttons
set Bcount 0
while {[gets $Conf Cline] >= 0} {
  if {1 == [string match #* $Cline]} continue
  if {[string length $Cline] < 4} continue
  set Nend [string wordend $Cline 0]
  incr Nend -1
  set Bname [string range $Cline 0 $Nend]
  set Cbeg [expr $Nend + 2]
  set Bcomd "exec "
  append Bcomd [string range $Cline $Cbeg end]
  incr Bcount
  set NextBut "button$Bcount"
  button .$NextBut -text $Bname -command $Bcomd
}
if {$Bcount == 1} {
  puts stderr "No buttons defined"
  return 2
}
# display buttons
while {$Bcount >= 1} {
  set NextBut "button$Bcount"
  pack .$NextBut -padx 10 -pady 10
  incr Bcount -1
}
button .exit -text Exit -command {exit}
pack .exit -padx 10 -pady 10
```

Section 5. Getting to know Expect

What is Expect?

Expect is an extension of the Tcl and Tk languages. Expect provides a simple, yet powerful interface for automating the scripting of interactive programs. Additionally, Expect makes it easy to embed interactive applications in a GUI. Expect's development is coincident with the emergence of Tcl/Tk, and depends on both in its current version, 5.32.

The author of Expect is Don Libes, who works at the US National Institute of Standards and Technology (NIST). The *home page for Expect* is hosted on NIST servers. (However, Expect and any related commercial or non-commercial products are explicitly not endorsed by NIST.) In the panels that follow, we'll look at a few examples of Expect scripts culled from the source code example directory, along with a brief overview of its command syntax.

Why learn something about Expect? To quote from Don's paper, "Using *expect* to Automate System Administration Tasks" (USENIX LISA Conference, Oct. 1990) "...the result is that the UNIX system administrator's toolbox is filled with representatives of some of the worst user interfaces ever seen. While only a complete redesign will help all of these problems, **expect** can be used to address a great many of them."

```
#!/usr/local/bin/expect --
# ftp-rfc <rfc-number>
# ftp-rfc -index
# retrieves an rfc (or the index) from ddn1
exp_version -exit 5.0
if {$argc!=1} {
  send_user "usage: ftp-rfc \[#] \[-index]\n"
  exit
}
set file "rfc$argv.Z"
set timeout 60
spawn ftp ftp.uu.net
expect "Name*:"
send "anonymous\r"
expect "Password:"
send "bilbrey@orbdesigns.com\r"
expect "ftp>"
send "binary\r"
expect "ftp>"
```

RFC retrieval with Expect

As an introduction to Expect, examine the example on the left. It is only slightly modified from the version found in the example directory from a standard Expect source distribution, as are all of the examples in this section. Let's walk through the code...

This program automates FTP retrieval of IETF RFC (Request For Comment) documents from the UUNet archive. The first line of the script invokes an Expect shell. Note that I've given the full pathname to the executable. That's safest, since it's hard to know the path environment of any given

```
send "cd inet/rfc\r"
expect "550*ftp>" exit
"250*ftp>"
send "get $file\r"
expect "550*ftp>" exit
"200*226*ftp>"
close
wait
send_user "\nuncompressing file - wait...\n"
exec uncompress $file
```

user. The script first checks for the Expect version, then prints a usage message unless the correct number of arguments is given.

Next, a *timeout* value is set to prevent the Expect script from locking up system resources if the FTP session spawned on the following line fails to connect properly. Most of the balance of the script are sets of *expect* / *send* command pairs. Each *expect* command waits for the specified output from the spawned program (ftp, in this case), and then *send*s the correct response. Note that there are two traps for ftp error codes following the *cd* and get instructions. In each case, the error code 550 is matched against first, and if true, then the script exits. Otherwise, following a 250 code (indicating success), expect drops through into the next command.

After the document is received, the script issues a *close* command to the ftp session. The wait command holds up script processing until ftp is terminated. Finally the script sends a message to the user, decompresses the downloaded RFC (or rfc-index), then exits by default, not explicitly.

```
#!../expect -f
# wrapper to make passwd(1) be non-interactive
# username is passed as 1st arg, passwd as 2nd
set password [lindex $argv 1]
spawn passwd [lindex $argv 0]
expect "password:"
send "$password\r"
expect "password:"
send "$password\r"
expect eof
```

The keys to Expect, Part 1

There are four key commands in Expect (the language, with an uppercase 'E'). First is *expect* (the command, little 'e'), which searches for patterns and executes commands if a match is made. For each *expect* command, there can be several groups, each composed of option flags, a pattern to match against, and a command or body of commands to execute. *expect* "listens" to SDTOUT and STDERR by default, until a match is made or the *timeout* expires.

Patterns are matched by default using Tcl's string

match facility, which implements globbing, similar to C-shell pattern matching. A `-re` flag invokes regexp matching, and `-ex` indicates that the match should be exact, with no wildcard or variable expansion. Other optional flags for *expect* include `-i` to indicate which spawned process to monitor and `-nocase`, which forces process output to lowercase prior to matching. For complete specifications, type **`man expect`** at a command prompt to view the system manual page documentation for Expect.

The second important command is *send*, which is used to generate input for the process being monitored by the Expect script. *send* incorporates options for sending to a specified spawned process (`-i`), sending slowly (`-s`, so as not to overrun a buffer, in serial communications for instance) and several others.

The keys to Expect, Part 2

On the left is a script called `carpal`, yet another example from the source Expect distribution.

spawn is the Expect command that's used to create a new process. It's appeared in every

```
#!/usr/local/bin/expect
# Script to enforce a 10 minute break
# every half hour from typing -
# Written for someone (Uwe Hollerbach)
# with Carpal Tunnel Syndrome.
# If you type for more than 20 minutes
# straight, the script rings the bell
# after every character until you take
# a 10 minute break.
# Author: Don Libes, NIST
# Date: Feb 26, '95
spawn $env(SHELL)
# set start and stop times
set start [clock seconds]
set stop [clock seconds]
# typing and break, in seconds
set typing 1200
set notyping 600
interact -nobuffer -re . {
  set now [clock seconds]
  if {$now-$stop > $notyping} {
    set start [clock seconds]
  } elseif {$now-$start > $typing} {
    send_user "\007"
  }
  set stop [clock seconds]
}
```

example we've used. At left, it pulls the path to the default shell executable, and spawns a new instance. In doing so, *spawn* returns a process ID, set in the variable `spawn_id`. This can be saved and set in the script, giving expect process control capabilities.

interact is the command that Expect uses to open communications between a user and a spawned process. The `-nobuffer` flag sends characters that match the pattern directly to the user. `-re` tells *interact* to use the following pattern as a standard regular expression, and the '.' is the pattern, matching each character as it comes in. While in interactive mode, Expect's redirection of the STDOUT and STDERR streams is also given back to the user by default.

What you can do with Expect

When your script invokes an interactive program, by default Expect intercepts all input and output (STDIN, STDOUT, and STDERR). This allows Expect to search for patterns that match the output of the program, and send input to the spawned process to simulate user interaction. Additionally, Expect can pass control of a process to a user if so instructed, or take control back upon request.

Not only do these traits make Expect remarkably useful for common administrative tasks, but it turns out that Expect is quite good for building test scripts to perform I/O validation during program development.

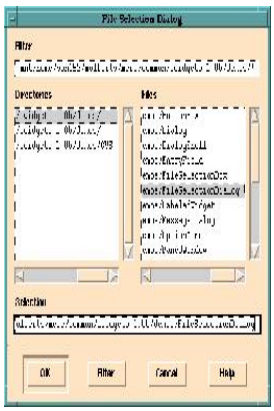
Finally, there's the stunningly useful program, `autoexpect`. Itself an Expect script, `autoexpect` monitors a command line interactive program, generating an Expect script that replicates that interaction exactly. Now, while that usually isn't just what's needed, it is easy to take the results of several autoexpect sessions, generalize the expect patterns, then cut and paste them into the desired configuration. It's been written in more than one place that the best learning tool for Expect is to run autoexpect and play with the results.

Section 6. Tcl/Tk extensions

Tcl/Tk extensions intro

Expect was merely the first arrival in a flood of Tcl/Tk extensions. Several are of general utility, many more are either software or application specific. As Scriptics remains the central repository for all things Tcl/Tk, the *Extensions Page* on that site is a valuable resource for further exploration.

In the panels that follow, we'll take a brief glance at several of the major extensions, touching on their salient features and attractions.



[incr Tcl], [incr Tk], and more...

Introduced in 1993, (pronounced *inker tickle*) provides object oriented functionality to Tcl/Tk. [incr Tcl] provides object, class, and namespace features. These features make it easier to build large projects with Tcl with data encapsulation, composition, and inheritance. This is done with the following: *classname*, *objname*, and *delete* are object commands. Class creation and editing is done with the *body*, *class*, and *configbody* commands. Other miscellaneous commands are *code*, *ensemble*, *find*, *local*, and *scope*.

Its graphical counterpart is . This tool extends to the GUI the same OO functionality that's needed to provide ease of scalability and data hiding that makes large programming jobs much easier to partition. *[incr Tk]* provides new base classes: *itk::Archetype*, *itk::Widget*, and *itk::Toplevel*. These classes are complemented with a complete set of methods.

Built on the *[incr Tk]* foundation, now there's , a so-called Mega-Widget set. This tool permits complex objects, such as a file selection box, to be defined and displayed with great ease. Borrowed from the *[incr Widgets]* Web pages, the image on the left is created using just the following commands commands: **`fileselectiondialog .fsd ; .fsd activate`**.

Tix

Tix, which stands for **T**k **I**nterface e**X**tension, is an extraordinary GUI and graphics Tcl/Tk extension. Tix is currently at version 4.0, and provides a set of 43 commands, most of which are either Mega-widgets or components for building Mega-widgets, along with a few utilities. Tix's Web site, at *http://tix.mne.com/*, claims "With Tix, you can forget about the frivolous details of the TK widgets and concentrate on solving your problems at hand." It's easy to see the basis of this claim, when you can quickly create useful interfaces with commands like *tixDirList*, *tixFileSelectDialog*, *tixPopupMenu*, *tixScrolledWindow*, and many more.

TclX

Extended Tcl, **TclX**, is really more than just another "extension". In the words of the authors, "Extended Tcl is oriented towards systems programming tasks and large application development. TclX provides additional interfaces to the native operating system, as well as many new programming constructs, text manipulation tools, and debugging capabilities." The online home of TclX is found at *http://www.neosoft.com/TclX/*.

Many of TclX's original features have made it into the core Tcl distribution over the last few years. However, the TclX team has stayed on the ball, adding such features as dynamically loading libraries and packages, network programming support, procedures that provide command access to the math functions normally called by expr, and much more.

TclX is included as a package that can be optionally installed with most standard Linux distributions. Alternatively, it can be compiled from source, in conjunction with Tcl and Tk. One very nice feature of recent TclX versions is a program called **`tclhelp`**, which is a Tcl and Tk help browser that is very handy to have around for reference. Highly recommended.

Visit the extensions emporium

Not unexpectedly for such a popular programming environment, there are many more extensions than I've space to discuss exhaustively in this tutorial. Point your browser to the *Extensions page* at Scriptics, and learn more about the following capabilities that Tcl and Tk extensions bring to the table:

* *Database extensions* permit access to everything from dBase files to connection and query toolkits. Many other database-related tools and utilities are also available.
* *Networking extensions* add functionality for email, FTP, network monitoring, distributed processing, Palm connectivity utilities, etc.
* *Object Systems extensions* include *[incr Tcl]*, and several other OO Tcl

implementations. These include OTcl (the MIT Object Tcl), XOTcl, TOS, Tea (a Java-styled Tcl OO), stooop (Simple Tcl-Only Object Oriented Programming), and more.

* Along with *[incr Widgets]* and the other Mega-widget libraries we've already seen, the *Tk Widgets extensions* provide a vast array of added GUI functionality, from image and video handling widgets to notebook and document interface extensions. This is definitely one of the places to check before you design something for yourself.
* The *Miscellaneous extensions* category is a complete grab-bag, with a little something for everyone. From an alpha toolkit for implementing neural networks (LANE), to data conversion, message digest and crypto packages (Trf and TrfCrypto), to audio play and record capabilities (Snack).
* Additionally, there are Tcl/Tk extensions explicitly for both the *Mac* and *Windows* implementations. If you work in either of these environments, you owe yourself a visit.

Section 7. Resources and feedback

Collected resources

Here is the concentrated list of Tcl/Tk resources for easy reference. First, most of the Web links found in this tutorial are at (or only a hop or two away from): *http://dev.scriptics.com/*, the home of Tcl/Tk development.

To complement those dynamic sources, there are a number of excellent books on Tcl/Tk and the extensions thereto. I can only recommend what I've read and used; you can browse at your favorite brick and mortar or online bookseller for any of the more than 20 titles.

* *Tcl and the Tk Toolkit*, by John K. Ousterhout (Addison Wesley: 1994, ISBN 020163337X). Although it's a bit long in the tooth, this book remains an excellent introduction to the style and practice of Tcl/Tk programming.
* *Practical Programming in Tcl and Tk, Third Edition*, by Brent B. Welch (Prentice Hall: 1994, ISBN 0130220280). Large, comprehensive, and full of clearly explained examples, I find this book constantly useful.
* *TCL/Tk Tools*, by Mark Harrison, et al. (O'Reilly and Associates: 1997, ISBN 1565922182). A great complement to the first two book listed, Tools is a multi-author effort that effectively covers many of the most popular Tcl/Tk extensions in enough detail to provide a clear benefit over the Web and manpage resources for some of these packages.
* Finally, there's *Tcl/Tk in a Nutshell*, by Paul Raines and Jeff Tranter (O'Reilly and Associates: 1999, ISBN 1565924339). All the commands are there, with flags, options, and

more for the Tcl and Tk core commands, the Tcl/Tk C interface, Expect, and several other major extensions. When there's a deadline looming, this is one handy book to have around.

Your feedback

We look forward to getting your feedback on this tutorial and for future directions in providing up-to-the-minute information about the always-evolving Linux scripting languages. Also, you are welcome to contact the author directly at *bilbrey@orbdesigns.com*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.