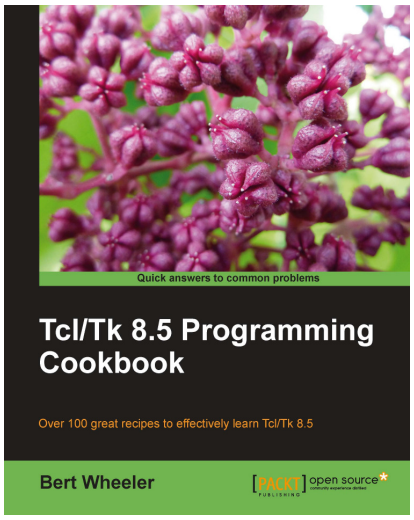
[PACKT] open source
community experience distilled
PUBLISHING

Tcl/Tk 8.5 Programming Cookbook

Bert Wheeler



Chapter No. 1
"The Tcl Shell"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "The Tcl Shell"

A synopsis of the book's content

Information on where to buy this book

About the Author

Bert Wheeler was born and raised in Louisville, Kentucky and is one of five sons. After graduating from high school he entered the United States Navy and later retired in 1997, after 20 years as an Air Traffic Controller. Following his military career, Bert returned to college and graduated Magna Cum Laude with a degree in computer science.

After completion of his degree, Bert worked in all aspects of the IT and software industries in numerous positions including Software Design and Development, Project and Product Management, Director of Information Technologies, and Director of Engineering Services. He continues to work and his primary area of expertise is in the design and implementation of physical security solutions in the Access Control arena throughout the world.

Tcl/Tk 8.5 Programming Cookbook

Created in 1988 by John Ousterhoult, while working at the University of California, Berkeley, Tcl (Tool Command Language) is a scripting language originally designed for embedded system platforms. Since its creation, Tcl has grown far beyond its original design with numerous expansions and additions (such as the graphical Took Kit or Tk) to become a full-featured scripted programming language capable of creating elegant, cross-platform solutions.

This book is written for both the beginning developer looking for a instructions on how to get their application up and running quickly to the experienced Tcl/Tk programmer looking to sharpen their skills. You will find everything from utilization of the console commands through to the creation of a stand-alone application.

What This Book Covers

Chapter 1, The Tcl Shell, gives an introduction to the Tcl shell.

Chapter 2, Using the Building Blocks Control Constructs, talks about using control constructs (if statements, for statements, and so on) to perform control program flow.

Chapter 3, Error Handling, talks about using the built-in commands and the Tcl shell to perform error handling.

Chapter 4, Handling String Expressions, explains how to create, manipulate, and manage string variables.

Chapter 5, Expanding String Functionality Using List, shows how to create, manipulate, and manage data in Tcl lists.

Chapter 6, The Tcl Dictionary, explains how to create, manipulate, and manage data in Tcl dictionaries.

Chapter 7, File Operations, tells how to open, read, write, and configure access to files stored on the system.

Chapter 8, Tk GUI Programming with Tcl/Tk, gives an introduction to the Tk shell, creating and managing a widget or window.

Chapter 9, Configuring and Controlling Tk Widgets, explains about creating and managing the most commonly used Tk widgets.

Chapter 10, Geometry Management, talks about managing the layout and design of the window.

Chapter 11, Using Tcl Built-In Dialog Windows, is about the creation and use of the Tcl built-in dialog windows available in Tk.

Chapter 12, Creating and Managing Menus, explains creating and managing menus, menu buttons, and pop-up menus.

Chapter 13, Creating the Address Book Application, gives full code listing and description of an Address Book application that makes use of the information covered in the previous sections.

1
The Tcl Shell

In this chapter, we will cover the following topics:

- The Tcl shell
- Writing to the Tcl console
- Mathematical expressions
- Tcl `expr` operands
- Tcl `expr` operators
- Mathematical functions
- Computing mathematical expressions
- Referencing files in Tcl
- Variables
- Launching a Tcl script

Introduction

So, you've installed Tcl, written some scripts, and now you're ready to get a deeper understanding of Tcl and all that it has to offer. So, why are we starting with the shell when it is the most basic tool in the Tcl toolbox?

When I started using Tcl I needed to rapidly deliver a **Graphical User Interface** (**GUI**) to display a video from the IP-based network cameras. The solution had to run on Windows and Linux and it could not be browser-based due to the end user's security concerns. The client needed it quickly and our sales team had, as usual, committed to a delivery date without speaking to the developer in advance. So, with the requirement document in hand, I researched the open source tools available at the time and Tcl/Tk was the only language that met the challenge.

The original solution quickly evolved into a full-featured IP Video Security system with the ability to record and display historic video as well as providing the ability to attach to live video feeds from the cameras. Next search capabilities were added to review the stored video and a method to navigate to specific dates and times. The final version included configuring advanced recording settings such as resolution, color levels, frame rate, and variable speed playback. All was accomplished with Tcl.

Due to the time constraints, I was not able get a full appreciation of the capabilities of the shell. I saw it as a basic tool to interact with the interpreter to run commands and access the file system. When I had the time, I returned to the shell and realized just how valuable a tool it is and the many capabilities I had failed to make use of.

When used to its fullest, the shell provides much more that an interface to the Tcl interpreter, especially in the early stages of the development process. Need to isolate and test a procedure in a program? Need a quick debugging tool? Need real-time notification of the values stored in a variable? The Tcl shell is the place to go.

Since then, I have learned countless uses for the shell that would not only have sped up the development process, but also saved me several headaches in debugging the GUI and video collection. I relied on numerous dialog boxes to pop up values or turned to writing debugging information to error logs. While this was an excellent way to get what I needed, I could have minimized the overhead in terms of coding by simply relying on the shell to display the desired information in the early stages.

While dialog windows and error logs are irreplaceable, I now add in quick debugging by using the commands the shell has to offer. If something isn't proceeding as expected, I drop in a command to write to standard out and voila! I have my answer. The shell continues to provide me with a reliable method to isolate issues with a minimum investment of time.

The Tcl shell

The **Tcl Shell** (**Tclsh**) provides an interface to the Tcl interpreter that accepts commands from both standard input and text files. Much like the Windows Command Line or Linux Terminal, the Tcl shell allows a developer to rapidly invoke a command and observe the return value or error messages in standard output. The shell differs based on the Operating System in use. For the Unix/Linux systems, this is the standard terminal console; while on a Windows system, the shell is launched separately via an executable.

If invoked with no arguments, the shell interface runs interactively, accepting commands from the native command line. The input line is demarked with a percent sign (%) with the prompt located at the start position. If the shell is invoked from the command line (Windows DOS or Unix/Linux terminal) and arguments are passed, the interpreter will accept the first as the filename to be read. Any additional arguments are processed as variables. The shell will run until the `exit` command is invoked or until it has reached the end of the text file.

When invoked with arguments, the shell sets several Tcl variables that may be accessed within your program, much like the C family of languages. These variables are:

Variable	Explanation
`argc`	This variable contains the number of arguments passed in with the exception of the script file name. A value of 0 is returned if no arguments were passed in.
`argv`	This variable contains a Tcl List with elements detailing the arguments passed in. An empty string is returned if no arguments were provided.
`argv0`	This variable contains the filename (if specified) or the name used to invoke the Tcl shell.
`TCL_interactive`	This variable contains a '1' if Tclsh is running in interactive mode, otherwise a '0' is contained.
`env`	The `env` variable is maintained automatically, as an array in Tcl and is created at startup to hold the environment variables on your system.

Writing to the Tcl console

The following recipe illustrates a basic command invocation. In this example, we will use the `puts` command to output a "Hello World" message to the console.

Getting ready

To complete the following example, launch your Tcl Shell as appropriate, based on your operating platform. For example, on Windows, you would launch the executable contained in the Tcl installation location within the `bin` directory, while on a Unix/Linux installation, you would enter `TCLsh` at the command line, provided this is the executable name for your particular system. To check the name, locate the executable in the `bin` directory of your installation.

How to do it...

Enter the following command:

```
% puts "Hello World"
Hello World
```

How it works...

As you can see, the `puts` command writes what it was passed as an argument to standard out. Although this is a basic "Hello World" recipe, you can easily see how this 'simple' command can be used for rapid tracking of the location within a procedure, where a problem may have arisen. Add in variable values and some error handling and you can rapidly isolate issues and correct them without the additional efforts of creating a Dialog Window or writing to an error log.

Mathematical expressions

The `expr` command is used to evaluate mathematical expressions. This command can address everything from simple addition and subtraction to advanced computations, such as sine and cosine. This eliminates the need to make system calls to perform advanced mathematical functions. The `expr` command evaluates the input and arguments, and returns an integer or floating-point value.

A Tcl expression consists of a combination of operators, operands, and parenthetical containers (parenthesis, braces, or brackets). There are no strict typing requirements, so any white space is stripped by the command automatically. Tcl supports non-numeric and string comparisons as well as Tcl-specific operators.

Tcl expr operands

Tcl operands are treated as integers, where feasible. They may be specified as decimal, binary (first two characters must be 0b), hexadecimal (first two characters must be 0x), or octal (first two characters must be 0o). Care should be taken when passing integers with a leading 0, for example 08, as the interpreter would evaluate 08 as an illegal octal value. If no integer formats are included, the command will evaluate the operand as a floating-point numeric value. For scientific notations, the character e (or E) is inserted as appropriate. If no numeric interpretation is feasible, the value will be evaluated as a string. In this case, the value must be enclosed within double quotes or braces. Please note that not all operands are accepted by all operators. To avoid inadvertent variable substitution, it is always best to enclose the operands within braces. For example, take a look at the following:

- `expr 1+1*3` will return a value of `4`.
- `expr (1+1)*3` will return a value of `6`.

Operands may be presented in any of the following:

Operand	Explanation
Numeric	Integer and floating-point values may be passed directly to the command.
Boolean	All standard Boolean values (true, false, yes, no, 0, or 1) are supported.
Tcl variable	All referenced variables (in Tcl, a variable is referenced using the `$` notation, for example, `myVariable` is a named variable, whereas `$myVariable` is the referenced variable).
Strings (in double quotes)	Strings contained within double quotes may be passed with no need to include backslash, variable, or command substitution, as these are handled automatically (see the chapter on *String Expressions and Handling* for clarification on these terms and their usage).
Strings (in braces)	Strings contained within braces will be used with no substitution.
Tcl commands	Tcl commands must be enclosed within square braces. The command will be executed and the mathematical function is performed on the return value.
Named functions	Functions, such as sine, cosine, and so on.

Tcl supports a subset of the C programming language math operators and treats them in the same manner and precedence. If a named function (such as sine) is encountered, `expr` automatically makes a call to the `mathfunc` namespace to minimize the syntax required to obtain the value.

Tcl `expr` operators may be specified as noted in the following table, in the descending order of precedence:

Operator	Explanation
- + ~ !	Unary minus, unary plus, bitwise NOT and logical NOT. Cannot be applied to string operands. Bit-wise NOT may be applied to only integers.
**	Exponentiation Numeric operands only.
*/ %	Multiply, divide, and remainder. Numeric operands only.
+ -	Add and subtract. Numeric operands only.

Operator	Explanation
<< >>	Left shift and right shift. Integer operands only. A right shift always propagates the sign bit.
< > <= >=	Boolean Less, Boolean Greater, Boolean Less Than or Equal To, Boolean Greater Than or Equal To (A value of 1 is returned if the condition is true, otherwise a 0 is returned). If utilized for strings, string comparison will be applied.
== !=	Boolean Equal and Boolean Not Equal (A value of 1 is returned if the condition is true, otherwise a 0 is returned).
`eq ne`	Boolean String Equal and Boolean String Not Equal (A value of 1 is returned if the condition is true, otherwise a 0 is returned). Any operand provided will be interpreted as a string.
`in ni`	List Containment and Negated List Containment (A value of 1 is returned if the condition is true, otherwise a 0 is returned). The first operand is treated as a string value, the second as a list.
&	Bitwise AND Integers only.
^	Bitwise Exclusive OR Integers only.
|	Bitwise OR Integers only.
&&	Logical `AND` (a value of 1 is returned if both operands are 0, otherwise a 1 is returned). Boolean and numeric (integer and floating-point) operands only.
x?y:z	`If-then-else` (if `x` evaluates to non-zero, then the return is the value of `y`, otherwise the value of `z` is returned). The `x` operand must have a Boolean or a numeric value.

Mathematical functions

Mathematical functions (such as sine and cosine) are replaced with a call to the Tcl `mathfunc` namespace. This does not require any additional syntax to access the namespace as it is called automatically. These are invoked by passing the Function followed by the value or values to evaluate to the expr command. Those functions that accept multiple arguments require that the arguments be comma delimited. The default Mathematical functions are listed below in alphabetical order. These functions require a specific syntax (for example `expr {function(value,value)}`) to be accessed, as described in the *Computing mathematical expressions* section that follows:

Function	Result
`abs arg`	Returns the absolute value of `arg`.
	Numeric operators may be integer or floating-point. Value is returned in the same format.
`acos arg`	Returns the arc cosine of `arg`.
`asin arg`	Returns the arc sine of `arg`.
`atan arg`	Returns the Arc Tangent of x/y.
`bool arg`	Returns the Boolean value of `arg` where non-numeric values are `true`, otherwise the value is `false`.
`ceil arg`	Returns the smallest floating-point integer value not less than `arg`.
	Any numeric value is acceptable.
`cos arg`	Returns the cosine of `arg`, measured in radians.
	If the result produces an over-flow, an error is returned.
`double arg`	Converts `arg` to its floating-point value.
	May return INF or –INF when the numeric value is such that it exceeds the floating-point value.
`entier arg`	Converts `arg` to its integer value.
`exp arg`	Returns the exponential of `arg`.
	If the result produces an over-flow, an error is returned.
`floor arg`	Returns the largest floating-point integer not greater than `arg`.
	The argument may be any numeric value.
`fmod x y`	Returns the remainder of `x/y` as a floating-point integer.
	If `y` is a zero (0), then an error is returned.
`hypot x y`	Returns the length of the hypotenuse of a right angled triangle.
`int arg`	Returns the low order bits of `arg` up to the machine word size.
`isqrt arg`	Returns the integer portion of the square root of `arg`.
	Arg must be a positive value (integer or floating-point).

Function	Result
`log arg`	Returns the natural logarithm of `arg`.
	`arg` must be a positive value.
`log10 arg`	Returns the base 10 logarithm of `arg`.
	`arg` must be a positive value.
`max arg`	This function accepts one or more numeric values and returns the greatest.
`min arg`	This function accepts one or more numeric and returns the least one.
`pow x y`	Returns the value of `x` raised to the power `y`.
	If `x` is zero (0), `y` must be an integer value.
`rand`	Returns a pseudo-random floating-point integer in the range of 0, 1.
`round arg`	Returns the rounded value of `arg` if `arg` is an integer value.
	If `arg` is not an integer, it is converted to an integer by rounding and the converted value is returned.
`sin arg`	Returns the sine of `arg` as radians.
`sinh arg`	Returns the hyperbolic sin of `arg`.
	If the result produces an over-flow, an error is returned.
`sqrt arg`	Returns the square root of `arg`.
	Accepts any non-negative numeric value.
	May return INF when the value is a numeric value that exceeds the square of the maximum value for the floating-point range.
`srand arg`	Resets the seed for the random number generator and returns a random number as described in `rand`.
`tan arg`	Returns the tangent of `arg` as radians.
`tanh arg`	Returns the hyperbolic tangent of `arg`.
`wide arg`	Returns the low order 64 bits of `arg`.
	Accepts any numeric value.

Computing mathematical expressions

In the following examples, we will see the correct syntax for both simple and complex mathematical formulas. To accomplish these computations, we will be using the Tcl `expr` command. The `expr` command, as its name implies, is used to evaluate mathematical expressions. This command can address everything from simple addition and subtraction to advanced computations such as sine and cosine. This removes the need to make system calls to perform advanced mathematical functions. The `expr` command evaluates the input and arguments and returns an integer, floating-point, or string value as appropriate.

A Tcl expression consists of a combination of operators, operands, and parenthetical containers (parenthesis, braces, or brackets). There are no strict typing requirements so any white space is stripped by the command automatically. Tcl supports non-numeric and string comparisons as well as Tcl specific operators.

As you will see, some computations may be performed without parenthetical notations; however, it is best to get into the habit of always using them. For example, `expr 1+1` and `expr (1+1)` will both return a value of `2`. While the omission of the parenthetical notation is completely acceptable in this usage of the `expr` command, I recommend developing the habit of always using them.

My personal favorite is the `if-then-else` expression. It provides a rapid method for comparison in a "single line" format. For example, if x and y are equal to 10, while z = 4 would be entered as `expr ($x?$y:$z)`. This expression evaluates `$x` as a Boolean expression. If it's true the expression will return `$y`; if it's false, it returns `$z`.

Parenthetical notation is required for any operation that will access a specific mathematical function. For example: `expr {pow (8, 4)}` will access the mathematical `power` function and return a value of 4096.

Variable substitution is handled using the Tcl `$` notation. The following example uses an `x` variable with a value of `4` and is entered as `expr {pow (8, $x)}`. This expression returns a value of 4096 as observed in the previous example. In the second case, `$x` has been processed with its variable value of `4`.

Referencing files in Tcl

Tcl commands that accept filenames as arguments require that they be in one of three formats, depending on the platform in use. The platform in use is stored in the global `TCL_platform` array variable, created at the start of the program. Please note that to address issues of portability, you must manually manipulate the formats to ensure that they are annotated correctly.

These formats are **absolute**, **relative**, and **volume-related**.

File Formats	Explanation
Absolute	Absolute names are fully qualified and give a path to the file relative to a particular volume.
Relative	Relative filenames are unqualified and give the path to the desired file relative to the current working directory.
Volume-related	Volume-related filenames are partially qualified and either accepts the path relative to the current working directory on the current volume, or relative to the directory of a specified directory.

The following conventions are platform-specific annotations for both the directory structure and the specific filenames.

UNIX (UNIX, Linux and Mac OS X)

On the UNIX style platforms, Tcl uses path names, wherein the various components are separated by the slash (/) character. Multiple adjacent slashes are handled as a single occurrence. Trailing slashes are ignored completely. For example, passwd and passwd/ both refer to the file passwd in the current directory

Convention	Meaning
`.`	Special character that refers to the current directory
`..`	Special character that refers to the parent directory
`/`	Root directory
`/etc/passwd`	Absolute path to the file passwd in the directory etc
`passwd`	Relative path to the file passwd in the current directory
`etc/passwd`	Relative path to the file passwd in the directory etc from the current working directory
`../passwd`	Relative path to the file passwd in the parent directory

Windows

Tcl supports both drive-related and **Universal Naming Convention** (**UNC**) file naming conventions. Both the slash (/) and backslash (\) characters may be used as separators; however, care must be exercised when utilizing the backslash characters, as they can result in undesirable effects *if* the filename is not enclosed within quotes. Drive-related filenames consist of the optional drive letter followed by the absolute or relative path. UNC filenames follow the form of `\\servername\sharename\path\file`. The UNC filename must contain the server and share components, at least.

Convention	Meaning
`.`	Special character that refers to the current directory
`..`	Special character that refers to the parent directory
`\\MyServer\MyShare\passwd`	Absolute UNC path to the file passwd on server MyServer in the share MyShare
`C:passwd`	Volume related path to the file passwd in the current directory
`C:\passwd`	Absolute path to the file passwd in the root directory of the C drive

Convention	Meaning
`\passwd`	Volume-related path to the file passwd in the root directory of the current volume.
`etc\passwd`	Volume-related path to the file passwd in the directory etc on the current volume.

In addition to the filename conventions listed in the preceding table, Tcl supports the Berkeley UNIX **C Shell** (**csh**) tilde (~) substitution. In the case of a filename with a preceding tilde, it will be interpreted by replacing the tilde with the current user's home directory. This is not platform-dependant.

Variables

As with all the programming languages, it is the variable that allows for true flexibility and usability. Tcl differs from some scripted languages, as, there is no need to implicitly declare the variable type. For example a variable of "3" will be stored within Tcl with the same internal representation, as if it have been defined as the integer 3. If the variable is then used in a calculation, Tcl will then convert it to an integer for computation. This is referred to as **shimmering** in Tcl.

Basic variable commands

Variable command	Explanation
`global var`	This command is used to declare a global variable. It is only required within the body of a procedure.
`incr var value`	This command will increment the value stored in `var` by the value provided. Value must contain an integer. If no value is passed, the command defaults to increase the value by one (1).
`set var value`	This command sets `var` to the value provided. Conversely, the value may contain a Tcl command, the results of which will be utilized as the final value. The Command must be enclosed within square braces.
`unset var var var`	The unset command deletes one or more variables. If the –`nocomplain` flag is passed as the first argument, all the errors are suppressed. Variable names are NOT comma delimited.

In the following examples, we will create a variable with an integer value of 3, increment that value, and then delete the variable.

Getting Ready

To complete the following examples, launch your Tcl Shell as appropriate, based on your operating platform.

How to do it...

For setting a variable, enter the following command:

```
% set x 3
3
```

How it works...

The set command returns `3` to confirm that the value was set correctly.

There's more...

Enter the following command:

```
% incr x 3
6
```

The `incr` command has increased the value of `x` by `3` and returned `6`.

Unsetting a variable

Enter the following command:

```
% unset x
%
```

The `unset` command deletes the variable `x` and simply returns to the command prompt.

If the named variable does not exist, an error will be generated, as shown in the following example:

```
% unset y
can't unset "y": no such variable
```

To avoid error reporting for variables, include the `-nocomplain` switch, as illustrated here:

```
% unset -nocomplain y
%
```

In this instance, the `unset` command has ignored the error and simply returned to the command line. This is invaluable when passing a list of variables to `unset` to ensure non-existing variables do not generate an error. Additionally, you should insert `--` (double minus, no spaces) after all the options, in order to remove a variable that has the same name as the many options.

Command line arguments

With any scripting language, the ability to provide arguments allows you to write a script that accepts arguments to perform a specific function.

As previously discussed, Tcl has several global variables to allow for the passing of command line arguments. The number of command line arguments to a Tcl script is passed as the global variable `argc`. The name of a Tcl script is passed to the script as the global variable `argv0`, and the arguments are passed as a list in the `argv` global variable.

Launching a Tcl script

In the following example we will invoke a Tcl script contained within a text file. This script will accept any number of arguments and print out the script name, the count of the arguments and the values contained within the `argv` variable.

Getting Ready

To complete the following example we will need to create a Tcl script file in your working directory. Open your text editor of choice and follow the instructions below.

How to do it...

Create a text file named `args.tcl` that contains the following commands.

```
# If no command line arguments are passed perform no actions
if {$argc > 0} {
# Print out the filename of the script
puts "The name of the script is: $argv0"
# Print out the count of the arguments passed
puts "Total count of arguments passed is: $argc"
# Print out a list of the arguments
puts "The arguments passed are: $argv"
# Using the List Index of argv print a specific argument
puts "The first argument passed was [lindex $argv 0]"
}
```

After you have created the file invoke the script with the following command line:

```
% Tclsh85 args.Tcl ONE 2 3
The name of the script is: args.Tcl
Total count of arguments passed is: 3
The arguments passed are: ONE 2 3
The first argument passed was ONE
%
```

How it works...

As you can see, the script accepts any number of arguments and using the Tcl global variables allows access to the arguments passed as either a list or individual values. Keep in mind that when passing control characters, they must be escaped using the backslash character.

There's more...

Invoke the script with the following command line:

```
% Tclsh85 args.Tcl \home \etc
The name of the script is: args.Tcl
Total count of arguments passed is: 2
The arguments passed are: home etc
The first argument passed was home
%
```

In the above example you can see that the backslash characters are removed. This is NOT done by Tcl, but rather by the shell from which Tcl was invoked.

Now invoke the script with the escape character added:

```
% Tclsh85 args.Tcl \\home \\etc
The name of the script is: args.Tcl
Total count of arguments passed is: 2
The arguments passed are: {\home} {\etc}
The first argument passed was \home
%
```

By adding the escape character the backslash characters are retained and curly braces have been appended to define the values as strings. For UNC file paths that contain double backslash characters you would need to enter one escape character for each backslash for a total of four. You may also 'protect' the data by enclosing it within quotes, however this is a feature of the shell used to invoke Tcl and not the Tcl shell.

Where to buy this book

You can buy Tcl/Tk 8.5 Programming Cookbook from the Packt Publishing website: `https://www.packtpub.com/tcl-tk-8-5-programming-cookbook/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/tcl-tk-8-5-programming-cookbook/book