

# Tcl Quick Introduction

## Introducing Tcl as Scripting Language for Design Tools and More ...

Brought to you by  
Dipl.-Ing. Martin Weitzel  
Technische Beratung für EDV  
<http://tbfe.de>

Inhouse Training for Rohde & Schwarz  
2019-01-21 + 2019-01-22  
Programming Logic Competence Center  
<http://plc2.com>

---

\*: You may use and copy the electronic version of this document freely as long as you comply with the [Creative Commons BY-SA License](#). As the presentation has been created with the free HTML4-Tool [Remark](#), its content is written using the [Markdown-Syntax](#). Therefore you may even enhance the purely electronic (non-printed) form with annotations only by means of an ordinary text editor. Just hit the P-key while viewing it in an internet browser and follow instructions.

# Agenda

---

1. The Tool Command Language
  2. The Programming Language Tcl
  3. The Tcl Standard Library
  4. Using Tcl in Vivado and Beyond
- 

Each part consists of some "theory" or background information and is enhanced by practical demonstrations, for which is left sufficient room in the time slot.

You are welcome to contribute – with your questions\* and also proposals what to try or which small changes to make, to see the outcome.

---

---

\*: Your questions will of course be answered to the best of the speaker's abilities ... and – on request – also in private communication during the breaks between the four parts.

# Part 1: The Tool Command Language

---

- A Look at Tcl's Internal Structure
  - Chances for Tools (like Vivado)
  - Tcl Limitations (for Tool Users)
  - Chances (not only) for Tcl-Aficionados
  - Tcl's Minimal Syntax
  - Variables and Subroutines (quickly\*)
  - Three Ways of Quoting
  - Syntax (Summary and Wrap-Up)
  - Trying and Understanding Tcl
- 

**Note:** All of the above will get practical coverage through the presentation. Any question is welcome, especially if it leads to varying the examples used (that way controlling how much emphasis is put onto which topic).

---

\*: This will be expanded in [Part 2](#).

# A Look at Tcl's Internal Structure

Understanding a little bit of Tcl's internal structure is helpful to get the big picture how Tcl is integrated into tools like Vivado.

- Basically all (textual) input undergoes [Syntax Analysis](#),
- including some relatively simple substitutions,
- the first word is looked-up in a table as name of a command,
- which is finally executed.

The command may be some Tcl subroutine – defined by the user or at startup of a tool in a configuration file – or it may be implemented in C/C++ and made available in a shared library (DLL).

## Chances for Tools (like Vivado)

Tcl lends itself perfectly to be extended by tools<sup>\*</sup>

- The look-up table consulted at the end of syntax analysis just needs to refer to tool-specific commands, in addition to (or instead of) what is built-in.
- Vivado uses this extensively, adding a huge number of Tool specific commands.

The latter are **not** the main topic of the following presentation, its focus is what Vivado users need to know to cope with Tcl in general.

---

<sup>\*</sup>: Not to much surprise, as this was an original goal of Tcl's initial design by [John Ousterhout](#).

## Vivado Tcl Documentation

For more information on the Tcl Vivado Integration see the following Vivado User Guides:

- [Using Tcl Scripting](#)
- [Tcl Command Reference](#)
- [Using Constraints](#)

The first – ug894 (~70 pages) – introduces into Tcl from the Vivado perspective, but is far from exhaustive with respect to Tcl.

The second – ug835 (1000+ pages) – is exhaustive with respect to the commands Vivado adds to Tcl, but most of it directly refers to the design objects (as represented in the internal database) on which the XILINX design flow for FPGAs is based.

The third – ug903 (172 pages total) – in some of its parts covers Tcl scripting in the special area of specifying design constraints.

## Running Tcl from Vivado

An interface to Tcl is provided by Vivado in several forms:

- By using the Tcl console provided by Vivado.
- By choosing an start-up option (like `-tcl` or `-batch`) and control every action of Vivado from Tcl.
- By switching between the Vivado GUI and Tcl Scripting with the commands:
  - `stop_gui` – issued from Vivado's Tcl console
  - `start_gui` – issued from Vivado's direct scripting mode

The Tcl console in the Vivado GUI has many useful options not provided in direct scripting mode, like extended syntax highlighting and context dependent completion and help features.

## User Communities

More information on Tcl is available in user communities like these:

- The Tclers Wiki – general Tcl (and Tk) topics <http://wiki.tcl.tk>
- The XILINX Tcl Store – with focus on using Tcl with XILINX products  
<http://www.xilinx.com/products/design-tools/vivado/Tcl-store.html>

Of course, you may also start a general search with your Tcl questions by using [Google](#) or any of its competitors, or visit less specific but usually well informed communities, like [StackOverflow](#).



## Chances (not only) for Tcl-Aficionados

As you have made it up to here, it is assumed you want to learn Tcl for making improved use of Vivado, so:

Why not use Tcl and possibly Tk for other purposes?

You could – e.g. – use the knowledge acquired some day also for

- writing small tools to automate recurring (... *stupidly repeating, terribly boring, mostly uninteresting* ...) tasks of your daily work,
- provide *easy to use interfaces* for users in technical environments, but with lesser affinity to scripting, or
- implement parts of the software for an embedded device based on a ZYNQ-board in Tcl(/Tk).

## Tcl Outside Vivado - Example 1

You need to debug, maintain and occasionally improve somewhat complex, automated Vivado design flow based in Tcl Scripting?

- Provide the necessary hooks on the Vivado side,
- instrument your Tcl script in Vivado accordingly,
- and add an external Tcl/Tk client to filter debug- and trace-information and further inspect what`s going on in Vivado.

Has been done – by the author of this\* ... and others probably too.

---

\*: With respect to the Tcl client – the application monitored and debugged was a C++ application with concurrently running C++ processes under the control of a supervising server, coordinating the TCP/IP interface.

## Tcl Outside Vivado - Example 2

For an embedded device with a touch small touch screen, running a Linux based application, implement the high-level control logic including in Tcl/Tk.

- Large parts of the development work can be done without access to the "real hardware":
  - Either on a hosted Linux Version (chose your favorite - Tcl/Tk is available not only on Linux but on most any U\*ix based OS, like BSD, Mac ...) ...
  - ... or on Windows (e.g. using the Tcl/Tk implementation freely available from [Active State]).
- You can show (and try-out) a working prototype early to your customer with the option to shift time critical parts closer to the hardware (Tcl → C/C++ → kernel driver or module → FPGA → dedicated hardware).

Has been applied – by the author of this\* ... and others probably too.

## Tcl Outside Vivado - Example 3

For an embedded device without any real user interface (except, maybe, some few, tiny buttons and Leds) but an Ethernet jack

- provide a socket based communication *from* the embedded device
- *to* some convenient control application running on a PC or Linux Host.

This can be done - and has been done - fully in Tcl/Tk by the author of this and others probably too.

You definitely need<sup>\*</sup> no expertise in any of the following areas:

- providing and configuring *apache* or some other web-server;
- using *PHP, Pearl, Python, ... whatever* on the embedded side;
- programming the control application in *JavaScript, Ruby, ... whatever*.

**To put it 100% clear:**

All of your code will be Tcl (when using TK for the GUI).

---

<sup>\*</sup>: Of course, you have also the *option* to use Tcl on the client-side only, or for the control application only, especially the latter, if you want to control your embedded device from every standard web browser ... but then, of course, you might have to learn a bit of HTML and JavaScript, at least.

## Tcl Limitations (for Tool Users)

The use of Tcl as command language in a tool also has a drawback.

- Every input first passes through Tcl's syntax analysis, especially with respect to the
  - the spelling of identifiers (e.g. permissible characters), and
  - the special meaning of some characters (like apostrophes, brackets ...)
- This may – in small areas – collide with the conventional use of the above in the domain specific use in other languages, tool users are accustomed to (like VHDL or Verilog).

As Tcl extensible in a number of ways, Vivado is even able to resolve some of these collisions, while others must be accepted by Vivado users, who have to learn how to avoid adverse effects.

# Tcl's Minimal Syntax

The Tcl syntax is minimal in various aspects – these are its main steps:

1. Line Concatenation
2. Command Separation
3. Word Separation
4. Substitutions of
  - Non-Printing Characters
  - Variables by Content
  - Subroutines by Return Value

Alongside the above, Several Kinds of Quoting are honoured:\*

- Backslash-Quoting
- Partial Quoting
- Full Quoting

---

\*: Each kind of quoting has its specific effects, as will be detailed soon.

## A Basic Example

The following example will be used in many variations for a detailed coverage of the Tcl syntax:

```
set greet "hello, world"
puts $greet
```

All output will be in a line of its own, i.e. puts automatically adds a newline at the end, unless ...

The output – of course – is:

```
hello, world
```

... it is requested **not** to do so:

```
puts -nonewline $greet
```

**From here on, please pay close attention to the presentation!**

In addition to the print-outs in your hands a lot more can be learned following the "live examples".

---

\*: Another difference, not relevant for the moment, is this: Any output to the screen is buffered and not actually visible until a newline is printed. Some consequences of this and how they can be avoided will be demonstrated later.

## Line Concatenation

Line concatenation takes place if a line ends with a backslash.

With the current knowledge of Tcl this can be demonstrated as following:

```
puts\  
$greet
```

```
puts \  
$greet
```

Looking closely to spot the difference between the command on the left and on the right should reveal that the newline character after the backslash is not completely purged but replaced by a space, explaining why all the following commands result in errors, but each for a different reason.

```
pu\  
ts $greet ;# attempt to call the non-existing command 'pu'
```

```
puts $\  
greet      ;# 'puts' gets two separate arguments, '$' and 'greet'
```

```
puts $g\  
reet       ;# attempt to substitute a variable named 'g'
```



## Command Separation

Next in the Tcl syntax analysis is looking for command separators.

- Besides an – unmasked(!) – newline
- also semicolons separate commands.

As a trivial example, the following prints a prompt and then gets some interactive input:

```
puts "start from count: "; flush stdout; gets stdin count
```

Usually in Tcl two commands are not written in the same line, unless they belong closely together and should not be accidentally separated.

```
puts -nonewline $greet; flush stdout      ;# show output unbuffered
```

---

\*: A realistic examples that makes it necessary to use flush is when the output of some program should become visible in small portions, but all in the same line, showing a "count-down":

```
while {[incr count -1] > 0} { after 1000; puts -nonewline "$count .. "; puts TAKING-OFF!
```

## Word Separation

Word separation has happened already in all the examples until, but may not have been noticed.

- Word separators are SPACE and TAB (in ASCII 0x20 and 0x09),
- written adjacently any number of the above is a single separator.

So all of the below equivalent:

```
puts $greet
```

```
puts    $greet
```

```
    puts $greet
```

```
puts    \
$greet
```

The option to insert more than one space\* were a single one is sufficient is rarely used in Tcl.

But sometimes the readability of regular and systematic code may improved by using columnar adjustments to align related parts.

---

\*: Using TAB instead of SPACE is discouraged or even banned in some style guides, because it usually depends on user preference set in the editor how they are expanded, hence code may look different for developers with different settings.

## Substitutions

There are several kinds of substitutions with quite different purpose<sup>\*</sup>

- Unprintable characters are substituted for some -sequences (basically the same set as in C/C++).
- Characters with a special meaning are substituted by themselves if preceded by a backslash (i.e. taken verbatim without being special).
- If \$ is followed by a "variable name" (composed from characters, digits, or underscore) the content of that variable is substituted.
- Any part of a command enclosed in square brackets will be taken as a (nested) Tcl command of its own:
  - it will be executed during the analysis of the embedding command, and
  - its return value will be inserted in place of the whole unit.

---

<sup>\*</sup>: In addition, line concatenation (already covered) may also be seen as a kind of substitution.

## Variables and Subroutines (quickly)

At that point, a quick (and not very complete) introduction to

- Variables and
- Subroutines

makes sense, as otherwise not many meaningful examples are possible

Both topics will get additional and deeper coverage in [Part 2](#).

For now, they are in kind of "cookbook-style", as suggested by the live examples.

## Variables (quickly)

Variables are **defined** and their content can be **substituted**.

To define a variable the Tcl command `set` is used:

- It can have one or two arguments:
- The first is the name of the variable.
- The second (if present) is the new value.
- It always returns the value (current or new).

To substitute the current value of a variable, its name is preceded by a dollar sign:

```
set greet "hello, world"
puts $greet
set greet "Guten Morgen"
puts $greet
```

Accessing variables that do not exist cause an error.

## Subroutines (quickly)

From the caller's perspective, a Tcl command can be anything:

- a built-in command implemented in C or C++ (provided by the Tcl core or the tool that uses Tcl as its command language);
- a subroutine implemented in Tcl,
  - either of from the [Tcl Standard Library](#)
  - or (again) supplied by a specific tool using Tcl as command language;
- a subroutine implemented in Tcl and previously defined by the same script that also calls it.

In any case the [Tcl Syntax Analysis](#) will identify **what is to call** by the **first word**, while **more words** are arguments and **handed over to the subroutine** to be used inside for any purpose<sup>\*</sup>

---

<sup>\*</sup>: Note that exactly that is an attractive feature of Tcl, as it allows for "incremental learning": the Tcl syntax itself is minimal and can be fully comprehended in little over an hour. Everything else to learn is on part of some command and hence depends what is required to solve a particular problem in hand.

## Pre-Defined Subroutines

As any existing Tcl command is a subroutine too, and called as any subroutine, the basics can be easily understood by what has been used so far: arguments and its return value can be inserted into another command line:

```
set x 10  ;# calling subroutine 'set' with arguments 'x' and '10'
           ;# to the effect that variable x is set to 10
incr x    ;# calling subroutine 'incr' with argument 'x'
           ;# to the effect that variable x is incremented by 1
incr x -5 ;# calling subroutine 'incr' with arguments 'x' and '-5'
           ;# to the effect that variable x is decremented by 5
puts $x   ;# calling subroutine 'puts' with an argument that is the
           ;# substituted content of variable x
```

## Using Command Substitution

To understand command substitution ([ ... ]) it is necessary to know what a subroutine returns:

- set and incr – as used the last page – return the new value assigned with by that operation;
- set with only one argument returns the current value of the variable;
- puts returns and empty string.

Knowing this, it should be easy to tell what will be the effect of the following commands (continuing from the last page):\*

```
puts [incr x]           ;# prints 7 (10+1-5+1) to the console
puts [set x]            ;# prints 7 (content of x) to the console
puts [set x 1]          ;# sets variable x to 1 and prints 1
                        ;# (new content of x) to the console
puts -nonewline [puts $x] ;# a contrived way to do 'puts $x' ...
puts [puts $x]          ;# ... as before, plus another newline
```

---

\*: Well, may be not *quite* easy in the last two examples, but feasible – the key point is to understand the difference between commands that print something on the console (puts) and the return value of commands (any command has one).



## Defining Subroutines

Subroutines are defined with the command `proc`,

- followed by the name that will later be used to call (execute) them,
- the list of their "formal parameters", and
- the Tcl code that contributes their "body".

As the topic gets its real coverage only in the next part, the following example does something rather unusual: It defines a command with the exotic name `=` which does nothing but print its arguments:\*

```
proc = {args} {  
    foreach arg $args {  
        puts "[incr n]:>$arg<:"  
    }  
}
```

---

\*: Note that up to Tcl version 8.4 it was necessary to set `n 0` zero before entering the `foreach` loop. Since Tcl 8.5 `incr` assumes for an unset variable it will start with zero.

## The Final Command Line

Understanding what is actually executed when Tcl executes a command – i.e. what the final command line looks like after syntax analysis, and when all substitutions are done – is crucial for understanding the Tcl syntax.

The subroutine defined on the last page can help here.

Try it as follows:\*

```
= set greet
= set $greet           ;# assuming greet is set to some value
= set greet whatever ;# this, of course, will NOT change greet ...
= puts [incr greet]   ;# ... OTH, this WILL change greet (why?)
```

---

\*: What makes an interesting experiment is to use = with its own definition:

```
= proc = args {
  set n 0
  foreach arg $args {
    puts "[incr n]:>$arg<:"
  }
}
```

## Intermezzo: "Funny Names" (1)

Unlike most any other programming language Tcl does not restrict what the name of a function or variable has to look like, because

- accessing variables and
- executing functions

is "just table lookup" only, where the name serves as key and therefore can be everything.\*

Except for rare cases it does not seem advisable to exploit that freedom.

Whether the subroutine from the last page falls into that category or not is disputable.

The "funny name" = was used for it to point out that it does nothing with a real purpose ... except to help Tcl-beginners to understand how a command looks **after** the Tcl Syntax Analysis is done and its substitutions are applied.

---

\*: Also note that procedures and variables with the same name may coexist, since they are looked up in different tables.

## Intermezzo: "Funny Names" (2)

As a general rule: **Avoid unusual names** – even if Tcl supports them – and stay with the rules of most high-level programming languages:

Compose name identifiers from letters, digits and underscore only.\*

Having said that, it might *sometimes* be convenient or even elegant to use "funny names", especially

- for **variables** that store kind of "internal secrets" and are not expected to be touched or modified often,
- for **subroutines** that do something very unusual or special, so the pure sight of the name should alert about this.

Hint: a nifty way to temporarily remove a subroutine **definition**, maybe to find out in which way other code depends on it, is to prepend a # to its name (not the proc command, this would only comment out one line).

---

\*: Besides readability this also is also necessary so that the content of variables can be easily expanded in the command line by prepending a \$ (dollar sign).. Variable names **not** only consisting of letters, digits and underscore need to be enclosed in curly braces when used after \$, while for subroutine no special care needs to be taken. And finally, as crazy as it may sound: it is even possible to use an empty string as name – just write an empty pair of curly braces or two adjacent double quotes where the name is expected.

# Three Ways of Quoting

There is one element in Tcl Syntax Analysis that has been ignored so far.

## Quoting

Basically quoting allows to hide parts of a command line from ordinary syntax analysis, so it shows up verbatim finally, when a command is executed and its arguments are handed over.

- Quoting with a (preceding) back-slash
- Quoting with (surrounding) double quotes
- Quoting with (surrounding) curly braces

Each one makes sense depending on the context and which parts of a command needs to outlive syntax analysis.

## Quoting with Backslashes

This is a variant and extension of what was already explained in the sections on [Line Continuation](#) and [Non-Printing Characters](#).

Any following character will be taken verbatim, i.e.

- the backslash is removed, and
- the character following is released from its special function,
- (assuming it has any, otherwise it simply stands as it is).

Note that the backslash may also be used to quote itself.

More will be demonstrated on the page showing [Quoting by Examples](#).

## Quoting with Double Quotes

Until the next **unquoted** double quote<sup>\*</sup>

- **no** newline or semicolon will be taken as **command separator**,
- **no** horizontal white space<sup>\*</sup> will be taken as **word separator**, and
- **no reduction of** adjacent **horizontal white space** to a single space takes place.

What still has its special function are

- **backslashes** (for producing non-printing characters and for quoting),
- **dollars signs** (for substituting the content of a variable), and
- **square brackets** (for evaluating what is inside as separate command and substituting the return value).

More will be demonstrated on the page showing [Quoting by Examples](#).

---

<sup>\*</sup>: Double quotes are what in German is known as *Gänsefüßchen*.

## Quoting with Curly Braces

It this way

- every character inside is released from its special function in the [Tcl Syntax Analysis](#),Q
- except that **contained** curly braces are counted to find the *final* closing brace, matching the *first* opening brace, and
- back-slashes do not change any content inside the curly braces, but may be used to **disable counting**, if used directly to the left of an (opening or closing) curly brace.

More will be demonstrated on the page showing [Quoting by Examples](#)



## Quoting by Example (1)

The best way to demonstrate the effects of quoting is with a subroutine which simply prints its arguments, like the one defined [here](#):

Try the following:

```
= puts "hello, world"  
= puts hello,\ world  
= puts {hello, world}  
= puts hello, world  
= puts ""  
= puts
```

The command puts - in the form used here\* - expects **exactly one** argument, which may be any string and will be printed verbatim on the console (followed by newline character).

Be sure to understand that puts receives

- exactly one argument in the first three examples, and
- **cannot see the difference in quoting(!)**,
- two arguments in the fourth example (so it were an error),
- one (empty string) argument in the fifth example, and
- no argument at all in the last example (so it were an error).

---

\*: Besides supplying the option -newline it is legal to use puts with two arguments, if the first one refers to an open file, as will be demonstrated in [Part 3](#).

## Quoting by Example (2)

Also try the following ...

```
= puts \n
= puts "  "
= puts hi! ;# ho!
= puts "hi! ;# ho!"
= puts {}
= puts {{{}}
= puts "{}"
= puts "\""
= puts \\
= puts "\\\"
= puts {\\"}
```

```
= { now see { that }! }
= { now see " that "! }
= " now see { that }! "
= " now see " that "! "
= {1" is 25,4mm}
= "1\" is 25,4mm}
= 1\" is\ 25,4mm
= {count {the} braces}
= {count {all {the}} braces}
= {count {all? the\\} braces}
= {count \{all? the\} braces}
```

... and request (many) more demonstrations – with explanations – if you think it helps to get a good understand of Tcl's quoting rules.

# Syntax (Summary and Wrap-Up)

As John Ousterhout states in his book "[Tcl and the Tk Toolkit](#)", some difficulties a Tcl novice may have stem from the assumption, the syntax must be more complex as it actually is.

Tcl-Syntax can be described very briefly as:

- Looking-up separators (for commands and words)
- Doing substitutions (variable content and command return values)
- All the way paying attention to quoting (\, " ... ", { ... })

Plus one more rule – may be an even more important one with respect to simplicity: **Do not repeat any of the above, once it is done.**

Therefore – and because looking up separators happens before variables and command return values are substituted – the following ...

```
proc say_goodbye {} { return "and thanks for all the fish" };  
puts [say_goodbye]
```

... hands **exactly one argument** to puts, not five.

# Trying and Understanding Tcl

From the author's point of view, one of the biggest advantages of Tcl over a compiled language like C or C++\* is this.

Most things in Tcl are easy to try – just start a `tclsh`\* and enter examples of the commands you want to understand.

The key word here is **understand**.

Trying "*to understand*" does not mean get something to work by using "*trial and error*" only!

- You may well try and vary and try again and vary again ... only
- **in the end** you should not any longer be surprised
  - **why it works**, and
  - **how it works**
- but be able to explain it to your colleagues, or at least to yourself.

---

\*: Or wish if you build a GUI based on Tk.

# Part 2: Tcl as Programming Language

---

- [Quick Syntax Summary](#)\*
  - [Essential Data Structures](#)
  - [Flow Control](#)
  - [Subroutines](#)
  - [Error Handling](#)
  - [Organising Reuse](#)
- 

**Note:** All of the above will get coverage through practical "live" examples during the presentation, and all attendees are invited to contribute proposals especially how to vary a certain example or what else to try – of course with the effects and final results always explained.

---

\*: This is just an *extremely brief recapitulation* of the section on [Tcl's Syntax](#) with back-links to the detailed coverage in [Part 1](#).

# Understanding the Syntax (quickly)

Why repeat anything in the era of electronic documents? We can link!

- For short summary of steps see [here](#).
- You will find more
  - about command separation [here](#) ...
  - ... and about word separation [here](#).
- Substitutions that eventually may occur are detailed
  - for non-printing characters [here](#),
  - for variable content [here](#), and
  - for subroutine return values [here](#).
- Finally you need to understand
  - quoting – see [here](#), with examples [here](#), and
  - that no step repeats, once completed – see [here](#).

# Essential Data Structures

Traditionally\* there are two important ways to structure data in Tcl:

- **Lists:**  
essentially sequential containers (of strings), numerically indexable with 0-origin (so rather like native arrays in C/C++)
- **Arrays:**  
essentially (string-based) look-up tables, aka. Hashes in some other scripting languages (so rather like `std::map<string, string>` in C++)
- **Dictionaries:**  
essentially "key-value" associations which may nest to any depth (unlike arrays which only allow a single level)

## Lists 101 - The Basics

Basically lists look similar to Tcl commands in that they may consist of any number of words:

```
set month_names "Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec"
```

The commands most often applied to lists\* are `llength` and `lindex`:

```
puts [llength $month_names]      ;# => 12
puts [lindex $month_names 0]     ;# => Jan
puts [lindex $month_names 1]     ;# => Feb
puts [lindex $month_names 11]    ;# => Dec
puts [lindex $month_names 12]    ;# => (empty string)
puts [lindex $month_names end]   ;# => Dec
puts [lindex $month_names end-1] ;# => Nov
```

Also, lists are often processed with `foreach`, fully introduced [later](#):

```
foreach month $month_names { puts $month }
```

---

\*: NB: "applied to lists" (i.e. the content), not "applied to list variables" (the name), therefore `$month_names`!



## Lists 101 - Nested Lists

If constructed properly, lists may also nest within each other, so with

```
set cities {Paris {New York} London Berlin}
```

the following list elements can be accessed:

```
puts [llength $cities]      ;# => 4
puts [lindex $cities 0]     ;# => Paris
puts [lindex $cities 1]     ;# => New York
puts [lindex $cities 2]     ;# => London
puts [lindex $cities 3]     ;# => Berlin
```

As the second element (with index 1) is itself a list, its parts can also be accessed individually:

```
puts [lindex [lindex $cities 1] 0] ;# => New
puts [lindex [lindex $cities 1] 1] ;# => York
puts [lindex [lindex $cities 1] end] ;# => York
puts [lindex [lindex $cities 2] 0] ;# => London
puts [lindex [lindex $cities 2] end] ;# => London
```

## Lists 101 - More on Nested Lists

It is also possible\* to apply several indices to select from nested lists in a single `lindex` command:

```
# same effect as commands on last page
...
puts [lindex $cities 1 0]      ;# => New
puts [lindex $cities 1 1]      ;# => York
puts [lindex $cities 1 end]     ;# => York
...
```

This will also work with the `lset` command to modify elements of a list:

```
lset cities 0 "City of Love"    ;# Paris => City of Love
lset cities 1 1 Amsterdam       ;# New York => New Amsterdam
```

**NB:** As the `lset` command **modifies** a list, a variable name must be specified as argument (`cities`), **not its content** (`$cities`).

---

\*: In more recent Tcl versions only, i.e. it was not originally a feature of Tcl, so you might not see it used in all applicable contexts.

## Lists 101 - Constructing Lists

Care has to be taken when lists are constructed from variables with arbitrary content.

The following will **not** work as (possibly) expected:

```
set capital_of_AR "Buenos Aires"  
set capital_of_DE Berlin  
set capital_of_GB London  
...  
set capitals "$capital_of_AR $capital_of_DE $capital_of_GB"
```

A more careful approach will use the command `list`, which **will** work as (probably) expected:\*

```
set capitals [list $capital_of_AR $capital_of_DE $capital_of_GB]
```

## Lists 101 - Extending (and Shorting) Lists

Appending to a list this should also always be done in a secure way:\*

```
set cities "$cities $other" ;# bound to fail for arbitrary content
lappend cities $other ;# always applies the "right" quoting
```

Finally there is the command `replace`, which not only allows to do what its name seems to suggest ...

```
puts [lreplace $cities end end Washington]
puts [lreplace $cities 1 2 Paris Madrid Lissabon]
```

... but also allows for removing content:

```
puts [lreplace $capitals 1 2]
```

Note that `lreplace` does **not** modify the content of a *list variable* but takes a *list* as argument and *returns* a modified list.

---

\*: At least if neither the content of the list, nor the value appended is predictable in that it has no "problematic" content. E.g. assume `other` may hold any value also something as weird as `"{\ }{'}"`.

## Arrays 101 - The Basics

Arrays are assumed by Tcl if a variable name follows a special pattern:

- After the regular name of a variable
- (without any space in between)
- follows a pair of round parentheses, holding the index.

An array index may be number ...

```
set month(1) January
set month(2) February
...
set month(12) December
```

... but also any string:

```
set days(January) 31
set days(February) "28 or 29"
...
set days(November) 30
set days(December) 31
```

Using the value follows the same pattern as setting it ...

```
puts $month(2)
puts $days(November)
```

... but may also assume more complex forms, as the index may also come from another variable:

```
set i 12
puts $month($i)
puts $month([incr i -1])
puts $days($month(2))
```

## Arrays 101 - Nested Indices and Multiple Dimensions

Actually indices can nest to any depth, which can be easily tested ...

... that way:

```
set i 1
set j(1) 2
set k(2) 3
set v(3) "here i am"

puts $v($k($j($i)))
```

Multiple array dimensions are not directly supported but may be easily simulated by

- writing two (or more) indices adjacent,
- separated with some unique character.

The only crux here is to find a separator for which it can be ensured that it will never be part of a valid index.

**But ambiguity here!**

```
set x 12
set y whatever
set v($x|$y)

set x left| ; set y right
set x left ; set y |right
```

## Arrays 101 - Accessing a Whole Array

The Tcl command `array` has several subcommands, all accessing a whole array in some way:

- `array size name` - returns number of entries in array
- `array names name` - returns list of all indices in array
- `array get name` - returns list of index-value pairs in array
- `array set name ...` - sets array from the list ... of index-value pairs\*
- ... (the above is not exhaustive) ...

Feel free to propose examples of the above be demonstrated live and further explained.

---

\*: What do you think: will this first purge the old content or just add new indices and values, in case the array does already exists with some entries?

## Dictionaries 101 - The Basics

Dictionaries are a more recent addition to the Tcl language and consistently managed via the `dict` command.

Grouped by function the subcommands are:

- Creating a dictionary: `dict create`
- Modifying a whole dictionary:
  - `dict append`
  - `dict lappend`
  - `dict merge`
  - `dict replace`
  - `dict update`
- Modifying single elements:
  - `dict set`
  - `dict incr`
  - `dict unset`
- Accessing single elements:
  - `dict get`
  - `dict exists`
- FP style dictionary processing:
  - `dict for`
  - `dict map`
  - `dict filter`
  - `dict values`
  - `dict with`
- Miscellaneous operations:
  - `dict size`
  - `dict keys`
  - `dict values`



## Dictionaries 101 - Creating a Dictionary

The following creates a dictionary with three elements and stores it in a variable named d:

```
set d [dict create A part1 B part2 SubSys {C Part3 D Part-4}]
```

- First element:
  - key: A
  - value: part1
- Second element:
  - key: B
  - value: part2
- Third element:
  - key: SubSys
  - value: C Part3 D Part-4

Be sure to understand the value of the third element  
**is itself a dictionary!**

## Dictionaries 101 - Accessing and Modifying Elements

Accessing top-level keys:

```
dict get $d A      # => part1
dict get $d B      # => part2
dict get $d SubSys # => C Part3 D Part-4
```

Accessing the nested level:

```
dict get $d SubSys C # => Part3
dict get $d SubSys D # => Part-4
```

Modifying a top-level key, a nested keys, and adding another nested key:

```
dict set d A Part-1
dict set d B Part-2
dict set d SubSys C Part-3
dict set d SubSys2 X Part-5
```

Note that `dict get $d ...` uses a dictionary variable **value**  
and `dict set d ...` uses a dictionary variable **name**.

## Dictionaries 101 - FP Style Dictionary Processing

A simple loop to print all top-level dictionary elements key-value pairs ...

... could be written like this ...

```
dict for {k v} $d {  
    puts "$k: $v"  
}
```

... to produce this output:

```
A: Part-1  
B: Part-2  
SubSys: C Part-3 D Part-4  
SubSys2: X Part-5
```

Applying a modification to all elements:

```
dict map {k v} $d {set v [string toupper $v]}
```

The dictionary then contains:\*

```
A PART-1 B PART-2 SubSys {C PART-3 D PART-4} SubSys2 {E PART-X}
```

---

\*: Note that while the top-level keys (SubSys and SubSys2) were not touched the nested dictionary's keys are just a part of the top-level key's value and **would now have been turned into upper-case now** if they hadn't been yet.

## Dictionaries 101 - Miscellaneous Dictionary Operations

Number of elements:

```
dict size $d          # => 4
dict size [dict get $d SubSys]  # => 2
dict size [dict get $d SubSys2] # => 1
```

Only the keys:

```
dict keys $d          # => A B SubSys SubSys2
dict keys [dict get $d SubSys]  # => C D
dict keys [dict get $d SubSys2] # => E
```

Only the values:

```
dict values $d          # => PART-1 PART2 {C PART-3 D PART-4} {E PART-X}
dict values [dict get $d SubSys]  # => PART-3 PART-4
dict values [dict get $d SubSys2] # => PART-X
```

# Flow Control

Flow control in Tcl allows for

- conditional execution (`if`)
- two-way or multi-way branches (`if-else` or `switch`), and
- repetition in various flavours (`while`, `for`, and `foreach`).

Before each of the above is demonstrated with examples, it is instructive what our `good ol' helper used to demonstrate quoting` will say here:

(assuming variables used are set)

```
= while {[incr i -1] > 0} {  
    # ... whatever ...  
}  
  
= if {$i} {  
    # ... whatever ...  
} else {  
    # ... whatever else ...  
}  
  
= for {set i 0} {$i < 10} {incr i} { puts -nonewline "$i .. " }  
  
= foreach m [array names $months] { puts $m }
```

## Intermezzo: `expr` and `eval`

There are two things, one that is required very often but Tcl does not support directly, and second Tcl does implicitly all the time good but which needs sometimes run explicitly.

- `expr` – **allows evaluation of arithmetic expressions**  
which the Tcl Syntax Analysis can not do directly but only through this command;
- `eval` – **evaluate an arbitrary string as commands**  
which the Tcl Syntax Analysis does all the time for with the scripts it executes but for flow control it needs to be available explicitly.

All flow control is **not** implemented as part of the Tcl Syntax but by separate commands.

In other words: If a Tcl script branches or repeats on some condition, from the perspective of the Tcl Syntax Analysis **always** some command gets started, which finally does what is expected.

## Evaluating Arithmetic Expressions

Only the command `expr` enables Tcl to evaluate arithmetic expressions, as demonstrated below (assume all variables used are appropriately set).

Some basic examples:

```
puts [expr 1+1]
puts [expr 1 + 1]
puts [expr {1 + 1}]
puts [expr {(7*12 - 18) / 32}]
puts [expr {1.0 / 2.0}]
```

Some more realistic usages.

```
set u [expr {$i + 2*$j}]
set v [expr {sin(0.66)}]
set w [expr {
    $x == 0 ? 0 : $y/$x
}]
```

Generally `expr` first concatenate all of its arguments to a single long string, which is then evaluated much similar to the expression syntax of C/C++.

Enclosing the argument in braces is not required but recommended, as it allows a more efficient evaluation in some contexts.\*

---

\*: The background is generating pseudo-code for a virtually machine Tcl uses internally, which has more chances for optimisation if Tcl variables as part of the arguments to `expr` are not evaluated at the "outer level" (i.e. variable substitution during syntax analysis) but at the "inner level" (inside the implementation of the command).

## Applying Bit Operations

As expr also supports the bit manipulation operations from C, any necessary low level processing is possible.

The following example counts how many bits are set in an integral value.

Assuming val is unsigned ...

... or assuming val has 32 bit:\*

```
set count 0
while {$val} {
    if {$val & 1} {
        incr count
    }
    set val [expr {$val >> 1}]
}
```

```
set count 0
set i 32
while {[incr i -1] >= 0} {
    if {$val & (1<<$i)} {
        incr count
    }
}
```



As right-shifting bits in a 2's complement integral number representation copies the sign bit, the example on the left side above may go in an endless loop for negative values.

---

\*: The internal representation of integral numbers in Tcl typically uses native types of the underlying hardware, so the details (size, behavior of right-shift, etc.) depends on the implementation.



## Evaluating Any String As Command

The command `eval` runs all the steps of [Syntax Analysis](#) on a string handed over as argument.\*

A typical example is storing a frequently used command – or a part thereof – in a string and then execute it:

```
# example from Vivado Tcl Reference Guide (UG835)
set runblocksOptDesignOpts { -sweep -retarget -propconst -remap }
eval opt_design $runblocksOptDesignOpts
```

**Without** the use of `eval` above

- command separators were looked-up only once,
- before the content of `runblocksOptDesignOpts` is substituted,
- therefore the blanks separating the options would not be properly recognized, and finally
- the Vivado command `opt_design` would complain about wrong usage.

---

\*: More exactly `eval` first concatenates all its arguments into one single string, then processes the result. It should be understood that using `eval` means there will be a **second pass** over the command line as originally spelled, "eating-up" one (more) level of quoting. Therefore, aside from the most simple cases, `eval` comes with its own set of potential pitfalls which must be carefully considered to be finally avoided.

## Branches with if (and if - else)

The Tcl command(!) `if` basically does the following:

- It hands over its **first** argument to `expr` for arithmetic evaluation, and – depending on the outcome –
- it hands over its **second** argument to `eval` for (further syntax analysis and) execution.

A condition is true if the arithmetic evaluation does result in some value different from zero.

Examples for the two most basic forms follow below:

Simple conditional execution, i.e. **one command block**, executed when the condition is true ...

```
if {$x < 2*$y} {  
    ... ;# condition true  
}
```

... and with **alternative command block** executed when the condition is false.

```
if {$x < 2*$y} {  
    ... ;# condition true  
} else {  
    ... ;# condition false  
}
```

## Chaining Branches with if - elseif

Again, to understand what is really behind branch chaining, it makes sense to look which arguments are really handed over to if:

```
= if {$x == $a} {  
  ... ;# x equal to a  
} elseif {$x == $b} {  
  ... ;# x equal to b  
} elseif {$x == $c} {  
  ... ;# x equal to c  
} else {  
  ... ;# neither of above  
}
```

Note that in the output (right) the arguments received are enumerated starting from 1.

I.e. from the viewpoint of if it are the 1<sup>st</sup>, 4<sup>th</sup> etc. arguments that hold the conditions (not the 2<sup>nd</sup>, 5<sup>th</sup> etc.).

```
1:>if<:  
2:>$x == $a<:  
3:>  
  ... ;# x equal to a  
<:  
:4>elseif<:  
:5>$x == $b<:  
:6>  
  ... ;# x equal to b  
<:  
:7>elseif<:  
:8>$x == $c<:  
:9>  
  ... ;# x equal to c  
<:  
:10>else<:  
:>  
  ... ;# neither of above  
<:
```

## Branches with `switch`

The command `switch` is an alternative to branch chaining with `if`.

Comparisons can be made in different ways, e.g. ...

... exact ...

```
switch -exact -- $x {
  $a {
    ... ;# x equal to a
  }
  $b {
    ... ;# x equal to b
  }
  $c {
    ... ;# x equal to c
  }
  default {
    ... ;# neither
  }
}
```

... or similar to typical file name  
pattern matching ...

```
switch -glob -- $f {
  *.bit {
    ... ;# bitstream file
  }
  *.vhdl |
  *.VHDL {
    ... ;# a vhdl file
  }
  *.vV {
    ... ;# verilog file
  }
}
```

... or with regular expressions, for which an [example follows](#) in [Part 3](#).

## Loops with `while`

Repeated execution of some code block with the command `while` is much similar to `if`:

- It hands over its **first** argument **repeatedly** to `expr` for arithmetic evaluation, and – depending on the outcome –
- it hands over its **second** argument to `eval` for (further syntax analysis and) execution.

The border case for `while` is no execution of the code block at all, if the condition evaluates to false from the start.

The following prints a count-down from 10, pausing for half a second after each value.

```
set count 10
while {$count > 0} {
    puts -nonewline $count; flush stdout
    after 500 ;# sleep 0.5 seconds
    incr count -1
}
```

## Loops with for

Using the command `for` instead of `while` usually improves readability of loops running through a consecutive range of values, as in this example:\*

```
# print 10 x 10 multiplication table
for {set i 1} {$i <= 10} {incr i} {
    for {set j 1} {$j <= 10} {incr j} {
        puts -nonewline [format "%3d " [expr {$i * $j}]]
        # puts + format is much like printf in C
    }
    puts ""
}
```

---

\*: Modified to use `while` the example would look like this:

```
# print 10 x 10 multiplication table
set i 1
while {$i <= 10} {
    set j 1
    while {$j <= 10} {
        puts -nonewline [format "%3d " [expr {$i * $j}]]
        incr $j
    }
    puts ""
    incr i
}
```

## Loops with foreach - The Basics

The command `foreach` specifically targets the processing of `Tcl Lists`.<sup>\*</sup>

Assuming a list `tm` of elements which in turn are sub-lists of two elements, holding a location and a temperature measurements, a detailed report ending with the average of all measurements may be printed as follows:

```
set sum 0.0
foreach measurement $tm {
    set loc [lindex $measurement 0]
    set temp [lindex $measurement 1]
    puts "temperature at $loc is $temp"
    set sum [expr {$sum + $temp}]
}
puts "average temperature is [expr {$sum / [llength $tm]}]"
```

<sup>\*</sup>: Most Tcl developers consider `foreach` more elegant and better readable, compared to this:

```
# processing a list with `for`:
set count [llength $tm]
set sum 0.0
for {set i 0} {$i < $count} {incr i} {
    set loc [lindex $tm $i 0]
    set temp [lindex $tm $i 1]
    set sum [expr {$sum + $temp}]
}
puts "average temperature is [expr {$sum / $count}]"
```

## Loops with foreach - Parallel List Traversal

A different form of `foreach` can be useful in a similar example, with locations and temperatures coming from different lists\*

```
set sum 0.0
set count 0
foreach loc $locations temp $temperatures {
    puts "temperature at $loc is $temp"
    set sum [expr {$sum + $temp}]
    incr count
}
puts "average temperature is [expr {$sum / $count}]"
```

The longer of both lists determines the number of runs through the loop. After the shorter list is exhausted, its placeholder variable is filled with an empty string inside the loop body.

---

\*: Though this data model seems inferior because the association of the different lists via an index only might be less robust, i.e. will more easily be broken.



## Loops with `foreach` and Arrays

Loops with `foreach` may be useful for `Tcl Arrays` too, as some `array` sub-commands return `Tcl Lists`.

The following example assumes temperature measurements stored in an array `tm`, indexed by location:\*

```
foreach location [array names tm] {  
    puts "temperature at $location is $tm($location)"  
}
```

Or with yet an alternate loop form:

```
foreach {temperature location} [array get tm] {  
    puts "temperature at $location is $temperature"  
}
```

---

\*: No code to calculate the average is provided here – it should be obvious how it would have to be added.

## Early Termination / Reevaluation

By using break loops need not run until a condition becomes false (or a list is exhausted), but can be prematurely terminated.

By using continue the loop body needs not run to its end but can branch back to reevaluate the condition (or extract the next list element, if any).

The following example calculates N primes:\*

```
set primes [list 2 3]
for {set next [lindex $primes end]} {[length $primes] < $N} {} {
    incr next 2
    set found 1
    foreach p $primes {
        if {$next % $p} continue
        set found 0
        break
    }
    if {$found} {
        lappend primes $next
    }
}
```

---

\*: Obviously a list of primal numbers is left in primes when after the loop terminates.

# Subroutines

Tcl subroutines are

- defined with the command `proc`
- taking the subroutine's name as first argument
- followed by a formal argument list, and
- the subroutine body.

The table associating names with the code to call is then extended by one more entry, branching to the subroutine body once the subroutine's name is recognised as command to execute, after syntax analysis is complete.

Formal arguments are the most complex parts (relatively) and will receive more coverage soon.

## Communicating through Global Variables

Packaging the [prime number calculation](#) into as subroutine requires little effort, if the communication takes place through global variables:

```
proc calculate_primes {} {  
    global N primes  
    ... ;# as before  
}  
...  
puts -nonewline "how many primes? "; flush stdout; gets stdin N  
calculate_primes  
puts "first $N primes: $primes"
```

Instead of naming the global variables in the command `global`, all references such variables may be preceded with two colons (`::`).

## Communicating through Argument- and Return-Values

An improvement over globals, which is easy to achieve, is the use of

- a (value) argument for handing over the number of primes to calculate, and
- the return value for transferring back the result.

That way none of the variables inside `calculate_primes` is visible to the caller and no variable belonging to the latter is reachable by the callee.

Packaging the [prime number calculation](#) into as subroutine requires little effort, if the communication takes place through global variables:

```
proc calculate_primes {N} {  
    ... ;# as before  
    return $primes  
}  
...  
puts -nonewline "how many primes? "; flush stdout; gets stdin cnt  
puts "first $cnt primes: [calculate_primes $cnt]"
```

## Communicating through Reference Arguments

For results yet another communication style is useful: reference arguments.

This requires

- on the *caller's side* to hand over a variable **name**, and
- on the *callee's side* to link that name via the command `upvar` with a local alias, effectively "reaching out" of the subroutine to the variable named by caller.

Accordingly modified the subroutine and its call will now look as follows:

```
proc calculate_primes {N vname} {
    upvar $vname primes
    ... ;# as before
    return [llength $primes]
}

...
puts -nonewline "how many primes? "; flush stdout; gets stdin cnt
calculate_primes $cnt result
puts "first $cnt primes: $result"
```

## Arguments with Default Values

The following example once more expects its first argument to be passed by reference (again applying upvar in cookbook-style), but also demonstrates a default value in case of its second argument.

In short, fincr extends what incr does to floating point variables:

```
proc fincr {vname {inc 1}} {  
    upvar $vname v  
    set v [expr {$v + $inc}]  
    return $v  
}
```

Some possible ways to use (and test) fincr are:

```
set x 1.5  
...  
incr x           ;# increments variable 'x' to 2.5  
incr x -0.5      ;# decrements variable 'x' to 2.0  
puts [fincr x 0] ;# prints 2.0 on console (variable 'x' unchanged)
```

## Variable Length Argument Lists

Variable length arguments lists have already been used in the helper function showing the command line after Tcl' syntax analysis has finished.

The essential mechanism is

- to specify name args as last in the formal parameter in the definition of a subroutine,
- what will cause to receive all arguments as a list – or those remaining after some fixed arguments, which must be always specified by the caller (see below).

The following example combines puts and format within a single function that behaves like printf in C, hence the name:

```
proc printf {fmt args} {  
    eval puts -nonewline {[format $fmt\n} $args {}}  
}
```

Make sure to understand why the implementation needs to use eval internally and also why the quoting is necessary.



## Details on `upvar` and `uplevel`

Most Tcl users apply `upvar` rather in "cookbook-style", like it was shown in the last example ... and that is usually sufficient.\*

Some Tcl users also know that there is a related command `uplevel`,

- much similar to `eval` in that it takes a string, applies Tcl syntax analysis and executes the resulting command,
- but arranges to make this happen **as if the caller** of the subroutine had executed that command.

The real use for this is to implement (new) ways of structuring control flow (which Tcl users rarely do), and some *block of code* has to be executed as if the caller had run it locally.

Do not worry if you have no idea what this page is about, once the time comes you need it, you will be experienced enough to understand it.

---

\*: As a rough sketch what goes on behind the scenes: each subroutine call has a *stack frame* where locals are stored. Arguments are locals in most ways sense, e.g. modifications are only applied locally and storage is space reclaimed once a subroutine ends. Arguments are special in one single aspect: the caller sets their initial value. The command `upvar` issued in a function does not exactly create a local variable but arranges that using a local name will actually access a variable in the *caller's* stack frame.

# Error Handling

If a subroutine cannot perform its advertised function, e.g. there may be bad argument values, it needs to indicate this to the caller.

Many programming languages provide a way to signal certain kinds of failure in special ways, so does Tcl with

- the command error that branches back ...
- ... possibly forcing a number of nested function calls to terminate ...
- to the next catch command, if any.

## Example: Using error

Using the command `error` is as simple as to call it after some failed test, with an argument describing the problem.

Again the subroutine to calculate primes is used to demonstrate this:\*

```
proc calculate_primes {N} {  
    # check if N is numeric  
    if {![string is integer $N]} {  
        error "argument is not a number: $N"  
    }  
    ... ;# as before  
}
```

Without further precautionary measures, calling `calculate_primes` with any non-numeric argument will now abort all further processing.

When the Tcl-Shell is used interactively, the above message will be produced and the user is returned to the command level.

---

\*: The command `string` used above to test whether `N` contains digits only will be covered in [Part 3](#).

## Example: Using `catch`

The command `catch` may be used to regain control after the command error has been executed.

The following example demonstrates how a program could be prompted repeatedly for interactive input, until the call to `calculate_primes` succeeds:

```
for {set done 0} {(!$done)} {} {  
    puts -nonewline "how many primes? "; flush stdout;  
    gets stdin cnt  
    if {[catch {calculate_primes $cnt} result]} {  
        puts "cannot calculate primes: $result"  
    } else {  
        puts "first $cnt primes: $result"  
        set done 1  
    }  
}
```

Another typical use pattern involving `catch` will be shown in [Part 3](#), when the handling of [errors on opening a file](#) is demonstrated.

# Organising Reuse

As soon as there are some 50 or 100 lines of Tcl code has been written, there may some reusable components emerge from it.

Basically there are three ways of organising reuse – assuming "*Copy & Paste*" of reusable components in the editor shall be avoided:

- Write subroutine definitions in a file of their own, and
- read this file explicitly with the command [source](#).
- Organise the files with reusable subroutine definitions in a way, so that [Tcl's Autoloading](#) will find and read it, whenever necessary.
- Organise your library components into [Tcl Packages](#).

All of the above can be subsumed under "advanced use" and in so far is no required knowledge to start using Tcl.\*

---

\* What may be useful to know for Vivado users is that **on startup** some files – presumably defining reusable subroutines – are looked up and read, if they exist. Such a file must be named `init.tcl` and be placed in either a sub-directory of the installation directory or of the user's home directory. (See [UG835](#) for details.)

## Part 3: The Tcl Standard Library

---

- [Syntax versus Library](#)\*
  - [Handling Character Strings](#)
  - [Formatting and Scanning](#)
  - [Using Regular Expressions](#)
  - [Date and Time](#)
  - [Working with Files](#)
  - [Command Line and Environment Variable](#)
  - [Running External Programs](#)
  - [Introspection \(and Debugging\)](#)
- 

**Note:** The above will also get coverage through practical "live" examples during the presentation. You are welcome to control with your questions and proposals for variations which parts will get increased attention which are only addressed cursory.

---

\*: This summarizes a few important points of [Part 1](#) and [Part 2](#) in an extremely condensed form, for all who choose not to attend to these, maybe because there is already some prior experience with Tcl programming.

# Syntax versus Library

As Tcl's syntax is only minimal, more of the learning effort needs to be invested on the side of the its standard library, but:

- This can be "lazily" and "incrementally":
  - Only the commands that need to be used need to be learned, and
  - as such are used promptly, experience builds up quickly.
- There are many systematic patterns,
  - so it is usually easy to "extrapolate" from existing experience,
  - though there are also irregularities ... occasionally.\*

---

\*: Most of these are due to backward compatibility and were rather due to, trade-offs, not breaking with the bulk of existing code. E.g. following the pattern emerging from later practice, all the commands handling lists should have rather been subcommands of a command `list`, but as it stands that command creates lists (only) and other purposes are served with other commands, all starting with an `l` (lower case letter ell).

## Learning Interactively

Nearly everything command in Tcl can be tried interactively

- and often as "one-liner" (i.e. without framing it into a larger program)
- so in one go with reading the manual any obscurities may be clarified.

A similar approach is used for demonstrating some important library functions during this part of the presentation.

Please feel free to interrupt the speaker to control the direction into which the live examples given evolve.

Let's start with looking up which standard commands are available:

`info commands`

Gives the whole list, but ...

... hmm, looks a bit unreadable, all these long lines without visual breaks ...

... better: with line breaks ...

```
join [info commands] \n
```

... even better: sorted!

```
join [lsort [info commands]] \n
```



# Handling Character Strings

The command `string` has a lot of subcommands. Some of the more frequently used ones are:

- `string length ...` – return number of characters\*
- `string index ... pos` – return single character at *pos*
- `string range ... from to` – return sub-string *from ... to* (**inclusive** range specification, so C++ STL users pay attention!)
- `string repeat ... count` – return *count* concatenations
- `string first what ...` – return position where *what* first matches
- `string last what ...` – return position where *what* last matches
- `string is class ...` – return if characters in strings are of *class*
- `string compare ... ..` – **attention** three-way compare (`== 0` is equal)
- `string equal ... ..` – compare exactly for equality (`!= 0` is equal)
- `string match pattern ...` – compare with *pattern* (according to file name pattern matching rules)

Many more are useful, like `toupper`, `tolower`, `trim`, `trimleft` `trimright`, ...

---

\*: Be aware of character set issues – with this and all other string handling commands. As Tcl internally uses 16 bit characters, Unicode with its whole [Basic Multilingual Plane \(BMP\)](#) will work flawlessly, but surrogate pairs may not, and – depending on the purpose for which the string is used – also combining characters might need special attention.

# Formatting and Scanning

The commands `format` and `scan` provide capabilities close to what `sprintf` and `sscanf` offer in C/C++.\*

The commands in Tcl work with strings of dynamic length.

C and C++ developers therefore

- may simply apply their existing knowledge ...
- ... using Tcl syntax,
- but without any worries about buffer overflows.

---

\*: Which is basically the same for strings as in C `fprintf` and `fscanf` provide for files, or `printf` and `scanf` for standard output and standar input.

## Formatting Example

Most often format is used if some output should be arranged in columns, like in the following table of Sine, Cosine, and Tangent functions:

```
set PI [expr {2*acos(0.0)}]
for {set angle 0} {$angle <= 360} {incr angle 15} {
    set line [format "%3d" $angle]
    set radians [expr {$PI*$angle/180.0}]
    foreach func [list sin cos tan] {
        set result [expr ${func}($radians)]
        if {abs($result) > 1e7} {
            set result "~inf"
        } else {
            set result [format "%.3f" $result]
        }
        append line [format " %11s" $result]
    }
    puts $line
}
```

## Scanning Example

Often scan is a useful helper to convert numeric strings into their value.\*

The following reads two floating point numbers and prints the diagonal, if both were sides of a rectangle:

```
while {[gets stdin line] > 0} {  
    set line [string trim $line]\n  
    if {[scan $line "%f%f%[\n]" x y dummy] != 3} {  
        puts "not two float values: $line"  
        continue  
    }  
    puts [format "diagonal of rectangle with sides a=%g and b=%g\  
                is c=%g" $x $y [expr {sqrt($x*$x + $y*$y)}]]  
}
```

Note that the example takes special care to recognize excess input after the second float, which would be silently accepted that way:

```
if {[scan $line "%f%f" x y] != 2} ...
```

---

\*: Despite the fact that this usually works automatically, there are occasions where more control is desirable, especially if "bad input" should be sorted out early.

## Reading and Writing Byte Values

Programming with Tcl close to the hardware sometimes requires to do numerical operation on byte values.

Here scan and format are useful too, as converting with the %c-specifier will translate between character values and (internal) integral numbers.

For some input stream – channel \$uart below, assumed to be in **translation mode binary(!)** – the following reads single bytes\*, then

- considers the upper Nibble as sequence number,
- the lower as four associated bit values, and
- assembles an array of sixteen elements,
- with each value being a list of four 0s or 1s

```
format [read $uart 1] %c byte
set index [expr {($byte >> 4) & 0x0F}]
set result($byte) [list]
for {set i 0} {$i < 4} {incr i} {
    lappend result($byte) [expr {($byte >> i) & 0x1}]
}
```

---

\*: The command read is covered in a [later section](#).

# Using Regular Expressions

Regular expressions offer elegant and powerful ways for processing character strings.

The main usage areas are:

- Very flexible string comparisons
- Extracting parts of a string
- Systematically modifying strings\*

The first two are handled with the command `regexp`, the last with `regsub`.

For most use cases it makes sense to enclose a regular expression in curly braces (`Tcl full quoting`).

The above avoids that any contained character gets a special handling in the `Tcl Syntax Analysis` and is eventually substituted by something else.

---

\*: In principle, as Tcl strings can have any content, regular expressions might also be used to manipulate binary data, though quoting must then be considered very carefully (inside and outside the regular expression) and specifying non-printable characters correctly comes with its own set of pitfalls, which even depends on the Tcl version used.

## Regular Expressions - The Basics

Basically there are

- **Atoms** – representing an inseparable unit, like
  - a specific single character, or
  - a character from a given selection,
- **Operators** – binary (postfix) and unary (infix),
- **Precedence Rules**, and
- **Round Parentheses** to alter precedence.

On the next slides follow a very brief (far from exhaustive) introduction to Tcl's Regular Expression Syntax and some basic usage examples.

For a detailed description of Tcl's regular expression syntax see the manual entry [re\\_syntax](#).

## Regular Expression Atoms

Most characters in regular expression are used literally, i.e. they stand as atoms for themselves. Notable exceptions are:

- `.` (dot) – represents any character
- `[...]` – represents any character listed inside (denoted here as ...)
- `[^...]` – represents any character **not** listed inside (denoted here as ...)

Furthermore a backslash allows to introduce the character following it as atom, and there are also escape sequences for non-printing characters.\*

- `\.` – represents a single dot;
- `[` and `]` – represent opening and closing square brackets;
- `\?`, `\*`, `\+`, and `|` – represent question mark, asterisk, plus, and vertical bar (which otherwise all are operators – see next page);
- `\\` – represents a backslash.
- `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, (and some more) – represent non-printing characters (like in C/C++).

---

\*: Be sure to understand that when it comes to regular expressions, there are actually **two** machineries in Tcl at work, that could cause to substitute things like `\n` or `\t` etc. by something else.



## Regular Expression Character Classes

A frequent necessity in a regular expression is to specify

- any *digit* – [0123456789] or [0-9],
- any *hex digit* – [0-9a-fA-F], or
- any *alphanumeric character* [0-9a-zA-Z],
- ... (etc.) ...



A range in square brackets, like a-z, means: Any *Code Point* in the underlying character set, from a to z inclusive.\*

As a solution a way to specify **Character Classes** is provided ...

- [:digit:] – decimal digits
- [:xdigit:] – hex digits
- [:alpha:] – letters
- [:space:] – white space
- ... (etc.) ...

... including **Shorthand Escapes**:

- \d – (decimal) digits
- \s – white space
- ... (etc., also inverted) ...
- \D – anything **but** digits
- \S – anything **but** white space
- ... (etc.) ...

---

\*: And therefore some examples above (especially a range like [a-z]) might not describe what is intended, which might also be the case with character classes with respect to localization.

## Regular Expression Operators

Regular expression operators are listed below in order of decreasing precedence (left box higher, right box lower, inside boxes top to down):

Postfix operators control repetition of atoms (or sub-expression) on their left side:

- $?$  means  $0...1$  (optional)
- $*$  means  $0...∞$  (any length)
- $+$  means  $1...∞$  (at least one)
- $\{m,n\}$  means  $m...n$
- $\{m\}$  means  $m$  exactly
- $\{m, \}$  means  $m$  at least

Infix operators connect atoms (or sub-expressions) on their left and right side:

- **Adjacency:**<sup>\*</sup>  
*first left then right hand side must occur.*
- **Separation by |** (bar):  
*either left or right hand side must occur.*

+

Parentheses may be used to modify precedence.

---

\*: Two adjacent sub-expressions may be read as joined with an "invisible operator" in between, i.e.  $ab$  as if it were  $a \cdot b$ , with  $\cdot$  has a precedence lower than all postfix operators and higher than the vertical bar.

## Regular Expression Simple Examples

Following are some simple examples for regular expressions.\*

Regular Expression ...	... and to what it matches
<[:alpha:]*>	sequence of letters in angle brackets
<.*>	... as before, not limited to letters
<[ ^<> ]*>	... as before, excluding angle brackets
\d+	non-empty digit-sequence (somewhere)
^\d+\$	... as before, but nothing else around it
-?\d+	integral number with an optional sign
[ -+ ]?\d+	... as before but allowing + <b>or</b> -
\d+\.\d*	signed integral or floating point number
\d+(\.\d*)?	... as before, in a slightly different way
\d+(\.\d*)?(e[ -+ ]?\d+)?	... as before, with optional exponent

---

\*: For subset of examples checking whether some string holds an integral or a floating number, note that this can be also achieved with `string is integer` or `string is double`, and in the latter case is much more complete with respect to what is recognised as valid float.



## Using Regular Expressions with `switch`

Regular expressions can also be used for comparisons with `switch`. The following assumes value may allow for various number formats (from binary to hexadecimal).\*

Parsing according to C/C++ ...

```
switch -regex -- $value {
{^0[1-7][0-7]*$} {
    ... ;# octal
}
{^-?[0-9]+$} {
    ... ;# (signed) decimal
}
{^0[xX][0-9a-fA-F]+$} {
    ... ;# hexadecimal
}
{^0b[01]+$} {
    ... ;# binary (since C++14)
}
```

... or more like VHDL ...

```
switch -regex -- $value {
{^X"[:xdigit:]+$}
    ... ;# binary (since C++14)
}
{^[01]+$}
    ... ;# binary (since C++14)
}
```

... (Verilog-ers will get their example on the next page ☺).

---

\*: The character classes specified as range also assume a machine character set with sequentially adjacent code points for 0 to 9 - which is guaranteed by the C/C++ ISO standard - and a to f and A to F - which holds for original (7-bit) ASCII, Extended (8-bit) ASCII, all ISO 8859 variants, EBCDIC and Unicode.

## Parsing with Regular Expressions

A regular expression also provides an easy way to access substrings that matched some part of it.

The following (more or less) recognizes binary literals\* from Verilog:

```
if {[regexp {(\d+)?([sS])?'[bB]([10zZ?]+)} $value\  
    match size signed bits}] {  
    ... ;# matching parts in  
    ... ;# - match : all (what matched)  
    ... ;# - size  : integral value or empty  
    ... ;# - signed: may be 's', 'S', or empty  
    ... ;# - bits  : non-empty sequence of '0', '1', 'z', 'Z', or '?'  
} else {  
    ... ;# did not match (variables not set)  
}
```

---

\*: A more involved example which allows for other bases is given, but without further explanations:

```
# first come the parts for the various number bases, they will then be connected via '|'
set based_values [list {[bB][01zZ?_]+} {[oO][0-7zZ?_]+} {[dD][0-9_]+} {[xX][0-9a-fA-FzZ?_]+}]
if {[regexp "(\d+)?([sS])?'([join $based_values |])" $value match size signed base_val]} {  
    set base [string index $base_val 0]  
    set val [string range $base_val 1 end]  
    ...  
}
```

# Date and Time

The command `clock` has a large number of subcommands, providing many operations with date and time.

These are the ones most often used:

- Current time as time-stamp in (milli/micro) seconds since epoch:  
`clock seconds`  
`clock milliseconds`  
`clock microseconds`
- Convert between time-stamp in seconds and human readable representation:  
`clock format`  
`clock scan`
- Calculations based on time-stamps and durations:  
`clock add`

# Working with Files

Working with files basically falls into three main categories:

- Working with Directories
- Operations with Files in Whole
- Accessing the Content of Files



## Working with Directories

The commands dealing with directories and directory content are:

- `pwd` – return the absolute path name of the current working directory;<sup>\*</sup>
- `cd` – change the current working directory according to the path name specified as argument or to the user's home directory, if no argument is given;
- `glob` – return list of path names in given or current directory (see examples following).



As it is a pre-process resource, be careful when changing the current working directory, as determines how relative file path names are interpreted for all parts of the running program.

---

<sup>\*</sup>: This is implemented as built-in command with the same name as the command a traditional U\*ix system provided for that purpose. Therefore `pwd` in Tcl not only works at the interactive interpreter level (where any unknown command is tried as operating system command anyway).

### Example for pwd and cd

The commands `pwd` and `cd` may be used in concert to remember and restore the current working directory.

The base technique is:

```
set old [pwd] ;# save current working directory
cd ...      ;# change working directory
...         ;# do work
cd $old     ;# restore old working directory
```

Note that the above is not very secure against failures.

Especially it will create problems if one of the command running while the directory is changed causes an error which is caught without restoring the working directory, the change will unexpectedly persist.\*

---

\*: The wrapping required to make saving and restoring secure against intermediate failure is not hard to write but also not trivial, so it is not shown here. If the feature to run a portion of a Tcl program with a different working directory is required more often, it will probably be best to implement a new control structure for that purpose with the help of [uplevel](#).

## Example for glob

The basic use of glob is trivial and easy, as it returns a list of path names for the current or a given directory.

All files matching \*:

```
... [glob *] ...
```

All files in directory /tmp:

```
... [glob /tmp/*] ...
```

There are many useful options, some are shown in the examples below:\*

```
... [glob -hidden *] ...    ;# include hidden file
... [glob .* *] ...        ;# as before on Unix or Linux
... [glob -- -*] ...        ;# files with names starting with '-'
... [glob -types d *] ...   ;# only (sub-) directories
```

Another important option is `-nocomplain`: when it is **not** specified it an empty result (list) will be considered as error.

---

\*: With respect to the example on "hidden" files it probably need to be understood that for Unix traditionally these are all files that have a name beginning with dot.

## Operations Files in Whole

The command `file` has a number of sub-commands for all the operations that are possible with a file in whole.

The subcommands are numerous and fall into the following categories:

- getting/changing file attributes\* (properties)  
file atime, file attributes, file exists, file isdirectory,  
file isfile, file lstat, file mtime, file owned, file readable,  
file readlink, file size, file stat, file type, file writeable
- copying, renaming, etc.  
file copy, file delete, file link, file mkdir, file rename,  
file tempfile, file volumes
- operations on path names  
file dirname, file extension, file join, file nativename,  
file normalize, file pathtype, file rootname, file separator,  
file split

---

\*: While in most cases results are delivered as return values `lstat` and `stat` expect the **name** of an array as argument and store the result at designated indices of that array.

## Accessing the Content of Files

Accessing the content of files from Tcl is modelled after C:

- open returns a file handle which usually is saved in a variable,
- as it is the entrance ticket for other operations with the file,
  - like reading or writing (read, gets and puts),
  - finding or setting the current position (tell and seek), and
  - miscellaneous operations like testing for EOF (eof)
- and finally handed over to close to release the resource associated with the file handle.

## Opening Files

The command `open` has two arguments:

- the file name and
- the open mode.

As usual, if the file name is a relative path name it is interpreted from the current working directory.

An unusual feature is that a path name beginning with a vertical bar (`|`) is considered to be external command. Depending on the open mode its input or output is then available via the file handle returned.\*

Open modes are specified

- either as for `fopen` in C (`r`, `w`, `rw`, `a`, ...)
- or in Posix style (`RDONLY`, `WRONLY`, `RWDR`, `APPEND`, ...)

Note that there are also some options effecting serial ports, but usually these are configure by `fconfigure`, which has still more options.

---

\*: This can be seen as a way of starting a process asynchronously and will be covered later

## Handling Errors when Opening Files

Errors on opening files with `open` are usually caught (and adequately handled) in an idiomatic style.

The following opens a file specified by variable `pathname` for reading:

```
if {[catch { open $pathname r } result]} {  
    ... ;# handle error (reason is in result)  
} else {  
    ... ;# work with file (handle is in result)  
    close $result ;# <--- essential to avoid resource leaks  
}
```

Note that closing a file – even if opened read-only – is usually necessary to avoid resource leaks.\*

---

\*: The problem may not show if only some few files are left in an opened state. But if a fragment like the above is run over and over again and does **not** close the file it opened, at some point the process may reach its limit of open files, and from that on **all** attempts to open a file will cause an error. On operating systems as used in the 1980s, this limit was about 20 or 50; recent OS may have raised it to several hundred, but on most any OS there still is such a limit.

## Deferred Handling of Errors when Opening Files

If an error from open cannot be sensibly handled **locally**, it is often easier to leave error to the caller of a function.

Return content of text file as Tcl list (each line one element):\*

```
proc get_file_lines {name} {
    set handle [open $name r]
    set content [read -nonewline $handle]
    close $handle
    return [split $content \n]
}
...
if {[catch { get_file_lines $pathname } file_lines]} {
    ... ;# problem opening (or reading?) the file
} else {
    ... ;# OK (lines as list elements in file_lines now)
}
```



If the above could not fail on open but on read a resource leak may occur as the opened file will not be closed.

\*: A counterpart subroutine `put_file_lines` is shown later.



### Intermezzo: What Exactly Does catch?

The last example may serve as reminder of an some unusual aspect in the use of the command `catch` that needs to be used for regaining control after an `error`:

- The **return value** of that command holds the information, whether the command given as *first* argument
  - lead to an error – then catch returns "true" as condition for `if`;
  - everything went ok – then catch returns "false" and the `else` part gets control.
- Depending on the former, the **second argument** of catch will
  - either hold the message returned from error,
  - or the result of the command executed (last).

## Special File Handling Options

The command `fconfigure` provides various sub-commands to control many aspects of files.

The following list is by no means exhaustive:

- how buffering is handled (none, by line, larger block, ...),
- translation modes (e.g. newline → carriage return + newline or vice versa),
- if a read will block as long as no data is available,<sup>\*</sup>

Note that using the non-blocking modes for reading data will often lead to busy waiting.

Asynchronous designs with read operations in call-backs, registered with the command `fileevent`, are usually a better option, **though not easily available in Vivado**.

---

<sup>\*</sup>: This is mainly used for "device files" representing to a serial line interface, TCP/IP-sockets, or U\*ix pipelines. In some cases it may also make sense for console input.

## Configuring Serial Devices

The command `fconfigure` also provides sub-commands to control the operation of serial hardware interfaces.

Again, the following is by no means exhaustive:

- set baud rate, bit-frame, parity ...
- ... behaviour of control lines ...
- ... flow control ...

In the standard Tcl manual pages the details are not documented with `fconfigure` but in section [Serial Communications](#) of `open`.

## Reading Whole Files or Fixed Size Portions

The command `read` either reads a number of given characters from a file, or a **whole file**.

The following fragment reads single characters with waiting (assuming a valid file-handle `fh`):

```
while {![eof $fh]}
  set ch [read $fh 1] ;# <-- will block if no data is available
  ... ;# process ch
}
```

## Reading Characters Without Waiting

As already mentioned, a file handle may be configured not to wait for data being available on read:

```
...
fconfigure $fh -blocking 0
while {![eof $fh]}
  set ch [read $fh 1] ;# <-- will not block if no data, so ...
  if {[string length ch] == 0} { ;# ... we need to check ...
    ... ;# ... and can do something useful here (REALLY useful)
  } else {
    ... ;# process ch
  }
}
...
```



Really useful means exactly that: **REALLY USEFUL**. Simply doing nothing when no data is available will lead to busy waiting and eats-up precious CPU time while waiting.\*

\*: The very least were to sleep a small amount of time without causing CPU load. This can be achieved with the command `after`, followed by a numeric argument that specifies the time to sleep in milliseconds, but this always requires trade-off between lowering the CPU load and sub-optimal latency.

## Reading Line By Line

The command `gets` reads single lines from a file.

In detail its behaviour depends on the arguments used in the call:

- With **only** a file-handle as argument it returns the next line read,
- With a file-handle **and** a variable name as argument it
  - returns the number of characters read, and
  - leaves the line content in the named variable.

Depending on the translation mode configured for the file end-of-line conventions will be handled transparently.

In the line read (and in the count returned by the second form) there is no end-of-line character contained.

## Examples Reading Line By Line

The following two examples append each line read from a file as new element to a list (assuming a valid file-handle fh):

```
set result [list]
while {[gets $fh line] >= 0} {
    lappend result $line
}
```

```
set result [list]
while {[eof $fh]} {
    lappend result [gets $fh]
}
```

Though both ways may look equivalent, they do actually are different for files in which the last line is terminated with end-of-line characters (as is usually the case).

The loop is exited after the last line (as it is probably intended).

The loop is **not** exited after the last line, as reading it has not (yet) set the end-of-file state – this will only happen with the **next** call to gets.\*

---

\*: But that call then returns an empty string, hence the result list has another (empty) entry at its end.

## Writing to Files

The command `puts` has been used often in this presentation, but always with only one argument (and sometimes the option `-newline` too).

If used with two arguments, the first must be a file handle, typically obtained from a call to `open`.

The following is the counterpart to the example subroutine

`get_file_lines`.\*

```
proc put_file_lines {name content} {  
    set handle [open $name w]  
    puts -newline $handle [join $content \n]  
    close $handle  
}
```

---

\*: Why does the call to `puts` require the use of the `-newline` option?



## More Operations with Open Files

The commands

- `eof`,
- `seek`, and
- `tell`

support more operations with files via file handles.\*

Live examples will be given if this topic is of special interest.

---

\*: As are usually obtained from `open`.

## Flushing and (finally) Closing Files

The command `close` is required to release any resources connected to a file handle.



When a file is `close`-d, also the last buffered output is flushed. Hence this command may return an error for files that are written.

Also note that `flush`-ing the buffer only happens from the perspective of the Tcl application. On modern operating systems (and even in the hardware of the storage system) there are usually additional levels of buffering.

Data may not have arrived on a reliable persistent storage media, only because the file to which is written has been `flush`-ed or `close`-d via its file handle.\*

# Command Line / Environment Variables

The calling context of a script may be accessed via [Tcl Global Variables](#).\*

- `argv` are the command line arguments (without the name of the command itself) as list;
- `argc` is the same as `llength $argv`;
- `argv0` holds the path name of the script executed (not the interpreter, i.e. **not** `tclsh` or `wish`);
- `env` is an array holding the environment variables (as can usually expected plus what has been individually set via `export` in the shell).

By using global variables with two leading semicolons, e.g. `::env` instead of `env`, the above also works as part of a subroutine, without introducing the variable name via the command [global](#).

---

\*: There are global variables for many more purposes, like checking for the version of Tcl (`tcl_version`) or Tk (`tk_version`), platform specific properties like the file path name separator (`/` on U\*ix and `\` on Windows) or byte order etc., and much, much more.

## Example for Accessing Command Line Arguments

The following prints a message on standard error stream that includes the program name and terminates the Tcl interpreter with a given return status:

```
proc die {message {exitstatus 127}} {  
    global argv0  
    puts stderr "$::argv0 [FATAL]: $message"  
    exit $exitstatus  
}
```



A call to `exit` will usually terminate a Tcl script at any point, hence **resources not cleaned up automatically** at the end of the process may leak ...

... but an application may chose to redefine the command `exit` and do something different.\*

---

\*: Especially applications that may run arbitrary, user supplied Tcl scripts – like Vivado – will typically chose to ignore `exit` or at least do all necessary cleanup and may also save any data important for to be available for recovery on a restart.

### Example for Accessing Environment Variables

The following lists all the environment variables and their value in sort order:

```
foreach v [lsort [array names ::env]] {  
    puts "$v=${::env($v)}  
}
```

Or only those with a XILINX\_-prefix:

```
foreach v [lsort [array names ::env XILINX_*]] {  
    puts "$v=${::env($v)}  
}
```

After modifying or adding to the global array env, the result is visible as (new) environment for a child processes, easily demonstrated as follows:

```
set env(MINE) whatever  
exec env | grep MINE
```

# Running External Programs

The command `exec` is the general interface to run separate processes.

Via a mix of Tcl and (U\*ix) Shell syntax it provides many variations as exemplified below (including a meaningful mix thereof):

- run the process synchronously;
- run the process asynchronously;
- run several processes pipelined;
- return standard output after completion ...
- ... optionally including standard error output
- ... and/or turn abnormal termination into a Tcl error;
- connect to the Tcl interpreter via file handles or sockets.\*

Besides its main purpose to open classic files, also the command `open` provides easy ways to run a separate process while either

- supplying its standard input or
- receiving its standard output.

---

\*: This makes more sense for asynchronously run processes and will often necessitate to receive data sent *from* the external program to the interpreter in callbacks registered via `fileevent`.

## Running Programs Synchronously

The `exec` command may wait on the external program to end, i.e. run it synchronously.

A typical fragment may look like this:

```
set xprog ... ;# program to execute (with arguments)
set result [exec $xprog] ;# execute and receive stdout and stderr
```

The command `exec` may also be used to start a pipeline of more than one program.

## Running Programs in Background

The `exec` command may start the external program in the background, i.e. run it asynchronously (aka. as demon).

A typical fragment may look like this:

```
set xprog ... ;# program to execute (with arguments)
exec $xprog & ;# execute (ignoring the result)
```

The above is used if it does not matter when and how the background process ended.

The return value of an `exec` command like above, which is the **PID** of the process, may also be saved for later use:

```
set pid [exec $xprog &]
```

If the command `exec` is used to start a pipeline of more than one program in the background, the return value is a list of all process ids.



## Controlling Programs in Background

A saved PID will typically be used to control the program running in the background in some way, most often to ...

... test whether it is still running ...      ... or to forcefully terminate it.

```
if [[catch {
    exec kill -0 $pid
}} {
    ... ;# process terminated
} else {
    ... ;# process still running
}

# first ask politely ...
exec kill -TERM $pid
# ... then wait ...
after 5000
# ... and no more mercy now:
exec kill -KILL $pid
```

Linux and many other modern U\*ix systems allow for finer grained interventions via the /proc file system.\*

---

\*: At any time the entries within it represent a live snapshot of all currently running processes as PID-named sub-directories. Their entries turn allow to obtain information on and get control over these processes. For details see: <https://www.kernel.org/doc/Documentation/filesystems/proc.txt>

## Sending Input to a Program

The `open` command may also be used to start a program as an external process while asynchronously supplying its standard input via a file handle.

A typical fragment may look like this:

```
set xprog ... ;# program to execute (with arguments)
set handle [open |$xprog w]
while { ... produce more data for xprog to process ... } {
    puts $handle ... ;# whatever
}
close $handle ;# will wait for xprog to terminate
```

## Receiving Output from a Program

The `open` command may also be used to start a program as an external process while asynchronously reading its standard output via the file handle returned.

A typical fragment may look like one of the following two fragments ...

... either lines of text ...

```
set xprog ...
set fh [open |$xprog r]
while {[gets $fh line] >= 0} {
    ... ;# process another line
}
close $fh
```

... or arbitrary (binary) data.

```
set xprog ...
fconfigure\
    [set fh [open |$xprog r]]\
    -translation binary
set data [read $fh]
close $fh
... ;# process all binary data
```

# Introspection (and Debugging)

The main interfaces required for introspection and debugging are available via the Tcl commands

- `info`,
- `trace`, and
- `rename`

## Obtaining Information (on Most Everything)

By various sub-commands the command `info` provides information many internals of the Tcl interpreter.

The following is an overview only and by no means exhaustive:

- `info vars` – returns a list of all variables (by default of the current scope, but as an optional pattern may follow, by using `info vars ::*` all global variables may be obtained too);
- `info exists varname` – returns true if a variable *varname* exists.
- `info commands` – returns a list of all commands currently known to the interpreter (includes but is usually much more than the next);
- `info procs` – returns a list of all currently known Tcl subroutines
- `info args procname` – returns a list of arguments (including defaults) as expected from subroutine *procname*;
- `info body procname` – returns the body (implementation) of subroutine *procname*;
- `info functions` – returns a list of all mathematical functions currently supported by `expr`.<sup>\*</sup>

---

<sup>\*</sup>: It is easily possible to extend the support for mathematical functions, for details see section [Math Functions](#) in the manual page of `expr`.

## Tracing Variable Access and Subroutine Execution

The various sub-commands of the command `trace` provide options to get call-backs when

- variables are accessed (read, written, or unset), or
- subroutines executed (entry, exit, or line-by-line trace).

Live examples will be given if this topic is of special interest.

## Renaming and Removing Commands

The command `rename` allows to

- change the name of an existing subroutine, or
- completely remove it (by renaming it to the empty string "").

Live examples will be given if this topic is of special interest.

## Part 4: Using Tcl in Vivado and Beyond

---

### Tcl in Vivado

- Understanding Design Objects
- Quoting Correctly Applied
- Special Square-Bracket Rules
- Design Object Name Matching
- More Uses of Tcl
- Problematic Areas

### Tcl beyond Vivado

- Calling C/C++-Modules
- TCP/IP-Communication
- GUI-Programming
- (and more - if time allows)

---

The (relative) weight of the major sections is controlled by attendees and may vary between "50:50" or "80:20", with the 80% on either side.

There will be a short poll now so that time can be appropriately allotted.



# Tcl in Vivado (Summarising)

Tcl is integrated in Vivado as its Tool Command Language.

In general Vivado's Tcl interpreter is full-featured\* but in some few special-purpose areas it has restrictions.

During start-up Vivado looks at various standard places to find and eventually read and execute (= source) files with Tcl commands.

- These files are a good place for a collection of private subroutines needed frequently in a Tcl-based design flow.
- If the collection grows, consider to split it into logical groups, put each group into a file of its own and source these from inside the files Vivado which is automatically reads at start-up.
- If your subroutine library grows ven more, consider to make use of [Autoloading](#).

---

\*: Currently the `tcl_version` used in Vivado is 8.5, i.e. just one minor release step from the current mainstream version of Tcl, which is 8.6.

# Understanding Design Objects

At its core Vivado holds a Design Model in a set of interconnected Design-Objects.\*

These objects are grouped into classes with a set of properties, which can be queried and modified in various ways with commands added to Tcl by Vivado for that purpose.

## **Design Objects are a new type from the viewpoint of Tcl.**

This type is the Vivado specific addition to Tcl's (internal) standard types like integral and floating numbers or character strings.

References to design objects are handled transparently by Tcl and need no special consideration when held in a Tcl variable or list.

---

\*: This is mainly derived from the documentation and should be rather taken as conceptual description, not as internal details of the actual implementation.

## Design Objects in Lists

There are many commands in Vivado that return design objects, often as list of design objects.\*

A design object (reference) can be

- added *to* a list *from* a variable or another list in which was held, and
- taken *from* a list in which it is held and assigned *to* variable.

Handing around a design object reference that way will keep the object itself fully intact, with all its properties.

The only thing that is made sure is that lists of design object (references) always hold objects of the same class.

---

\*: This is called an "object container" in the Vivado documentation and actually a new Tcl type too. But most of the commands a Tcl developer expects to use for lists are available for object containers and hence it can be said they have the "look & feel" of a Tcl list.

## Converting to and from Design Objects

If on occasion a Tcl (built-in) command actually needs a character string, the design objects

- will **not** be completely encoded into pure text form\* (somehow),
- instead **only** the NAME property of the design object will be used.

Vice versa the conversion *from* a name to a design object (reference) requires the use of a one of Vivado's get\_ commands.

Of course, design objects may be transformed into a textual description by means of report\_property -all ..., though the output then is primarily targeted to a human reader, not for direct further processing.

---

\*: This is what makes design objects different from the built-in types: the latter are always serializable to a pure text form, hence may be stored externally and retrieved with their original content. For design objects that would make little sense due to the many interconnections they may have with other objects, which in turn would then have to be serialised, or the originally serialises object were not of much use anymore.

# Quoting Correctly Applied

In VHDL identifiers may be contain and surrounded with back-slashes, e.g.:

- `\abc\def uvw\xyz\`

This, of course, needs some quoting in Tcl, but full quoting

- `{\abc\def uvw\xyz\}`

will fail in that particular case, due to the trailing back-slash.\*

.N[ Therefore the recommended solution is this:

- `"\\abc\\def uvw\\xyz\\"`

**Use double quotes and write all contained back-slashes twice.**

---

\* Though the trailing backslash will not be touched, the closing square bracket is not recognized as it is now exempt from brace counting.

## Special Square-Bracket Rules

Via the unknown procedure the Tcl syntax analysis determines what should happen in case of unknown command.

This allows a nifty trick in Vivado, to allow the use of VHDL signal vectors in their typical forms like `[1:16]`, `[5]`, or `[*]` without quoting the square brackets:

- The syntax analysis does nothing special here, it simply identifies the command to be executed as being spelled `1:16`, `5`, or `*`.
- If the unknown procedure detects that there is no built-in command with such a name, it simply returns it (as a string) `[1:16]`, `[5]`, or `[*]`.<sup>\*</sup>

Problem Solved!

---

<sup>\*</sup>: Though, really, really interesting and "unexpected" things may happen in a Vivado design flow based on Tcl if you have the boldness to defined a procedure with a funny name that happens to look like a signal vector you use, e.g.: ``proc 1:16 {} { return {[16:1]} }`

# Matching Design Objects Names

When design objects are returned as lists from the various `get_`-commands, it is often the case that their elements are selected via match-patterns for their names.

There are two matching styles:

- The default is pattern matching as with Tcl's command `string match` is used on the individual parts of an object's hierarchical name.
- If requested via the `-regex` option comparisons will be based on Tcl's `regular expressions`.

The former is further controlled by the `-hierarchical` option together with `current_instance`, and is often more convenient. The latter is much more powerful but careful attention must be paid to correct quoting.

## Default Matching of Design Objects Names

This is also known as "glob-style matching" and further controlled by

- the `current_instance` setting and
- the `-hierarchical` option.

The model is similar to navigating in a hierarchical file system, but not as close as it may seem at first glance:

- The (Vivado) commando `current_instance` is somewhat similar to setting the working directory in that it sets the *working instance*.
- Glob-style patterns **not** containing the hierarchical separator `/` (i.e. a slash, just as in Unix file systems) search from that instance.
- Glob-style patterns containing the hierarchical separator will search deeper down from the current instance.

Note that there is no way to refer to the top-level directly when the current instance is set elsewhere (i.e. like using absolute path names in file systems).



## Recursive (Glob-Style) Matching

With the `-hierarchical` option the search pattern must **not** contain the hierarchical separator and is applied in a recursive search down from the current instance to each element of the composed name.

The Vivado documentation ([UG835](#), [pg. 16](#)) explains the `-hierarchical` option ...

```
set pattern ... ;# pattern to match names
current_instance "" ;# set scope to top-level
set result [get_cells -hierarchical $pattern]
```

... with the manual approach that had to be used alternatively:\*

```
set result {}
foreach hcell [list "" A B A/a1 A/a2 B/b1 B/b2] {
    current_instance $hcell ;# move scope to $hcell
    set result [concat $result [get_cells $pattern]]
    current_instance ;# return scope to design top-level
}
```

---

\*: A B A/a1 A/a2 B/b1 B/b2 is the whole set of cell names used in their miniature example, and "" needs to be added so that the top-level itself is included.

## Regular Expressions to Match Design Objects Names

This kind of matching is much more powerful but careful attention must be paid to correct quoting.



*Trial and Error* with regular expressions in combination with quoting will more often lead to frustration than to success.\*

This does not mean you should avoid it – it rather means you should feel familiar with both, regular expressions and quoting, when using both in concert.

Live examples will be given if this topic is of special interest.

---

\*: This is especially true if (part of) the selection pattern comes from a variable. Though the Tcl syntax can be generally considered as very "regular" and hence easy to grasp (as there are few rules only that apply strictly with little or no exceptions) be sure to develop a good understanding for its intricacies.

## More Uses of Tcl in Vivado

Actually the Tcl language is used in some specialised areas of Vivado too.

- Custom Design Rule Checks (DRC):
  - In this some boiler-plate code is necessary.
  - For the general recipe to be followed see the Vivado documentation ([UG894](#), [pg. 28](#)).
- Design Constraints (XDC)
  - While many Vivado specific commands is available ...
  - ... substantial built-in Tcl commands are deliberately blocked.\*
  - For details see the Vivado documentation ([UG903](#), [Appendix A](#)),

The restrictions to Tcl commands only apply if constraints are specified in \*.xdc files. In \*.tcl files all the usual Tcl built-ins are available.

---

\*: The reason for not supporting the full set of Tcl built-in commands is that Vivado provides a way to interactively modify constraints specified in \*.xdc files.

## Problematic Areas

All commands that register call-backs are often pointless (though these commands are available)

- as there is no event loop running by default, and
- if one is started explicitly, it would block the return to GUI mode.

What *will* work is to establish a TCP/IP-socket connection between the Tcl interpreter running on behalf of Vivado and some external program.

An external program connected to the Tcl interpreter inside Vivado via TCP/IP **may or may not** itself be written in Tcl.

Nevertheless, by giving up event-driven style at least on the side of Vivado, some communication options are lost!

For one-way communication also a classic pipeline may be used and is easily established for **reading** or **writing** via the command open.

---

\*: Though it might even be possible to arrive at an integrated solution if sufficient work is invested, also with respect to future Vivado versions that may change the internal working, it seems wise to separate event-driven architectures from each other.

# Tcl beyond Vivado

Using Tcl beyond Vivado can have several faces:

- Extend Vivado with Tcl (and possibly Tk) in areas which are not properly prepared for it.\*
- Use Tcl for local scripting purposes, from small to medium to large to mission critical applications, possibly touching the core competences of a software development department or a whole company (has been done ... successfully).
- Use Tcl combined with Tk to provide Graphical User Interfaces.

---

\*: This is not meant as criticism with respect to Vivado, it only means that there are use possible case not foreseen, especially when the necessity arises to merge event loops.

# Calling C/C++-Modules

Extending Tcl with modules written in C offers big chances for rapid prototyping,

- doing the first sketches in a Tcl-only implementation,
  - either as "proof of concept", or
  - to determine what actually is required
- then identifying the parts that need substantial performance improvements, and
- finally rewriting these in C or C++.

This should work with Vivado too,<sup>\*</sup> but the author has not yet tried it.

---

<sup>\*</sup>: Of course, this makes sense only if Tcl is used heavily to script the Vivado work flow and there are – and provably – parts which substantially profit from a re-implementation in C or C++.

# TCP/IP-Communication

TCP/IP-Communication usually has two sides:\*

- The *Server Side* which usually needs some concurrency – or at least a "user experience" that "feels" like, as can be achieved in a event driven design with compact and efficient call-backs.
- The *Client Side* which may or may not need concurrency.

As with Vivado the event driven programming model – though thoroughly supported by Tcl – is difficult to exploit, Vivado can only be extended with Tcl to work as a simple form of TCP/IP-client.

A possible idea what that could be was sketched in Part 1.

Live examples will be given if this topic is of special interest.

---

\*: Though note, that the data stream as such is symmetrical, once a socket connection has been established.

# GUI-Programming

GUI-Programming – contributed to Tcl via Tk\* – is the major extension added to Tcl very early and partially as "Proof of Concept" to demonstrate the extensibility of Tcl.

Here is your very first TK application:

```
#!/usr/bin/wish
wm title . "Hello, World of Tk"
pack [button .b\
    -text "Do NOT klick here!\n(please)" \
    -background red \
    -foreground white \
    -command exit
]
```

---

\*: Actually *Tk* is the abbreviation for *Toolkit* – so, strangely, a very general name was chosen for a Tcl extension with a very specific purpose.



## Tk Pro's

- Easy to learn, once you know Tcl including its peculiarities.
- Relatively "lightweight", i.e. small memory footprint compared to later GUI libraries.
- Excellent reputation for its portability across operating systems, its robustness and long-term stability / backward compatibility.
- Encourages to use layout management via strategies, not fixed placements.
- Enough widgets for all basic tasks, including a (very powerful!) text area and a (rather basic) canvas.

## Tk Con's

- If someone does not like Tcl for any reason, probably hard(er) to learn.
- Not so many "nifty" widgets as more recent GUI libraries.
- Some pre-manufactured standard-dialogues are no shining example for usability.
- Can not cope with Qt (also for other reasons) ... but does Tk strive to?

Despite of all the above, **Tk is very well suited** for adding a GUI to an embedded device with a touch screen, especially if the device software is based on Linux.

---

\*: This becomes especially visible in the standard file selection dialogue Tk shows under Linux, which is, well, "in need to getting used to". Though, Linux also has so many different file managers that a "standard look & feel", how to present files from which a user can select, is hard to derive.