

Tcl Quickstart*

Introducing Tcl as Preparation to its Use as Vivado Command Language

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

Inhouse Training for SAF Tehnica
2018-02-20 ... 2018-02-22
for Programming Logic Competence Center
<http://plc2.com>

*: You may download this presentation from the [Author's Internet Site](#) for any use in compliance with the [Creative Commons BY-SA License](#). As it has been created using the free HTML4-Tool [Remark](#), its content is written using the [Markdown-Syntax](#), so you may even enhance the purely electronic (non-printed) form with your own annotations, only by means of any ordinary text editor. Just hit the P-key while viewing this in an internet browser, and follow instructions.

Agenda

1. Tcl in Vivado – The Basics
 2. The Programming Language Tcl
 3. The Tcl Standard Library
 4. The Tcl – Vivado Integration
 5. Tcl in Vivado – Beyond the Basics
-



Part 1: Tcl-Vivado-Integration – The Basics

- 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^{*}

- 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.

^{*}: 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>
<http://www.xilinx.com/products/design-tools/vivado/Tcl-store.html>
- 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":
 - On a hosted Linux Version chose your favorite Tcl/Tk (besides on Linux Tcl/Tk is freely available on most any U*ix based OS, like BSD, Mac, ...)
 - On Windows you may want to use 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, i.e. from Tcl → to C/C++ → to a driver or kernel module → to the FPGA → to 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^{*} 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).

^{*}: 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 and Arbitrary Unicode 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
```

The output – of course – is:

```
hello, world
```

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

... 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
```

Can you explain this?*

*: Looking closely to spot the difference between the command on the left and on the right should reveal that the newline character after the back-slash 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 presented up to here, but it may not have been noticed – probably as it seems "natural".

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

Therefore all the below equivalent:

```
puts $greet
...
puts      $greet
...
      puts $greet
...
puts      \
$greet
```

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

Though, in some cases 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^{*}

- 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.

^{*}: In addition, line concatenation (already covered) may also be seen as a kind of substitution.



Substituting Non-Printing Characters

Non-printing characters are substituted for certain backslash sequences:*

Escape	ASCII	Code Point	Effect
\a	BEL	7 / X'07	audible sound
\b	BS	8 / X'08	back space
\t	HT	9 / X'09	horizontal tab
\n	LF	10 / X'0A	line feed
\v	VT	11 / X'0B	vertical tab
\f	FF	12 / X'0C	form feed
\r	CR	13 / X'0D	carriage return

When such a value is used as "character output" the effect described above is the usual behavior of a classic terminal or printer.

In general rendering software may or may not exhibit that behavior.*

*: E.g. a pseudo-graphical representation of the value may be shown in a small box, the character may be replaced by a space, or even completely dropped.



Substituting Unicode Characters

Arbitrary Unicode Characters can be specified by their code point value (in octal or hexadecimal):

Notation		Unicode Range
\os	os = 1 to 3 octal digits	U+000000 ... U+0000FF
\uhs	hs = 1 or 2 hexadecimal digits	(as above)
\uhs	hs = 3 or 4 hexadecimal digits	U+000000 ... U+00FFFF
\Uhs	hs = 1 to 8 hexadecimal digits	U+000000 ... U+FFFFFF

When such a value is used as **Character Output** the glyph displayed in general depends on the **Stream Encoding** and should match the expectations of the rendering software.

Non-matching encodings will result in more or less "garbled" output.*

*: Completely unreadable output typically indicates that 8-bit and 16-bit encodings are mixed-up or the wrong byte order is chosen for a 16-bit encoding. Other mismatches may only show occasional glitches – e.g. **within** stream encodings like UTF-8 (for characters outside the range of 7-bit ASCII) or UTF-16 (for characters outside the basic multilingual plane).



Substituting Variables Content

The content of variables is substituted by prefixing the variable name with a dollar (\$):

Setting a variable:

```
set greet "hello, world"
```

Substituting the variable content:

```
puts $greet
```

While it is possible to use any character^{*} in a variable **name**, this is usually avoided as it would complicate substituting the value:

```
set "say hello" "Guten Morgen"
```

```
puts ${say hello}
```

Any quoting may be used for the variable in the set-command ...

```
set {say hello} "Guten Morgen"  
set say\ hello "Guten Morgen"
```

... but there is **no choice** when substituting the content (i.e. the curly braces are an extension of the \$ substitution syntax itself).

^{*}: Even the empty string is a legal variable name: set {} whatever; puts \${}



Substituting Returned Values

Enclosing a command in square brackets allows to

- use the **value returned** from the enclosed command (executed first)
- **as part of** another command (executed subsequently).

```
set oldpwd [pwd] ;# save current working directory, then ...  
cd $other      ;# ... change to other working directory ...  
...           ;# ... do some work there and finally ...  
cd $oldpwd     ;# ... restore saved working directory
```

Of course, the command enclosed in square brackets may also have arguments itself and use variable and nested command substitutions:*

```
set now [clock milliseconds] ;# get current time  
...           ;# (... do some work ...)  
puts [expr [clock milliseconds] - $now] ;# show elapsed time
```

*: But consider that temporary variables may make the code more readable:

```
set begin [clock milliseconds]; ...; set end [clock milliseconds]; puts [expr $end-$begin]
```



Order of Substitutions

All the substitutions described so far

- happen **in a strict order, one after the other**
- **not** repeatedly if the substituted value just "looks" special.

Consider ...

```
set dir {[pwd]} ;#1
...
... $dir ...    ;#2
```

Here – **because of quoting** –
command substitution **will not**
happen initially (at #1) ...

... but **will also not happen**
after variable substitution at any
later point in time (at #2).*

... compared to:

```
set dir [pwd] ;#3
...
... $dir ...   ;#4
...
... .. $dir ... ;#5
```

In this case command substitution
happens at #3 and consequently,
if variable substitution for `dir` is
requested several times, **that**
value is used (i.e. at #4 and #5).

*: Because, colloquially speaking, it's "too late" now.



Repeating Substitutions

Tcl's `eval` command runs a "second round of substitutions":*

```
set dir {[pwd]}      ;# actually stores `[pwd]` (including the
                      ;# square brackets) as content of `dir`
eval ... .. $dir ... ;# FIRST inserts `[pwd]` THEN runs second round
#    ^^-----^^--- CAUTION! TWO rounds of substitutions here!!!
```



Be sure to understand the consequences that for the `eval` command parts marked with "`^^^--`" in the last line above **two** rounds of substitutions take place.

Further substitutions on results of prior substitutions

- may or may not be an issue
- against precautions may have to be taken,

usually by some extra quoting:

```
eval {... ..} $dir {... ..} ;# NO first round of substitutions inside `{}`
```

*: This example does not intend to show a practical approach, it only continues the previous page.



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 can be set to a content which later can be **substituted** using their name. To define or change a variable the Tcl command `set` is used:

- The first argument names the variable.
- The second – if present – is a new value.
- Always returned is the value (existing or newly set).*

As [already has been shown in some more detail](#), to substitute the current value of a variable, its name is preceded by a dollar sign:

```
set greet "hello, world" ;# defines `greet` (if not yet existing)
puts $greet              ;# substitutes content of `greet`
set greet "Guten Morgen" ;# changes content of `greet`
puts $greet              ;# again substitutes content
```

Trying to access a variable that is not defined causes an error.

*: Accessing the content of a variable – say `x` – in a command substitution via `... [set x] ...` offers a second way to insert the content of `x` into a larger command line (besides the usual `... $x ...`). It may especially be considered for "indirect addressing", i.e. when the name of the variable to access is actually contained in some other variable – say `y`: Many consider `... [set $y] ...` to be more readable compared to `eval ... \$$y ...` – and less error prone too. (Why the latter?)



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.*

*: 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:

```
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 know what command substitution (`[...]`) inserts for pre-defined (standard) commands it is necessary to know what these commands return:

- `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 an 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 -newline [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 enumerate and print its arguments inside reversed angle brackets:^{*}

```
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"

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.*

*: An equals sign (=) was used as "funny name" for the command from the previous page to emphasize the point that its only purpose was to help Tcl-beginners to understand how a command **finally** looks, i.e. the Tcl Syntax Analysis is done and its substitutions are consequently applied.



Intermezzo: "Funny Names" - When to Avoid?

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.

- 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 \$.
- For names of subroutines no special care needs to be taken.

*: As crazy as it may sound: it is even possible to use an empty string as name!
To "use" it just write an empty pair of curly braces. (For a subroutine also two adjacent double quotes will do.)



Intermezzo: "Funny Names" - When to Use?

Having steered you away from "unusual" names possible in Tcl, it might *sometimes* be convenient or even elegant to still use them e.g.:

- For **variables** storing kind of "internal secrets" and are not expected to be touched or modified often,
- For **subroutines** doing something unusual or very special, so the pure appearance of the name should alert about this.
- To temporarily "mask" the definition of a subroutine.*

*: More concretely: there is a nice technique to temporarily remove a subroutine definition by prepending its name with a - sign. The motivation might be to find out which other code on this subroutine depends, because all its uses will then give an error message. (Some uses may then still be selectively re-enabled by prepending the - in the calling code.)



Three Ways of Quoting

There are yet three features of Tcl's Syntax Analysis that has been not covered in sufficient detail so far:

Quoting, Quoting, and 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 **should not be confused** with what has already been explained in the sections on [Line Continuation](#), [Non-Printing Characters](#), and [Unicode Character Substitution](#), only because the back-slash (\) is used for this feature too:

Any **other** character following a backslash will be taken verbatim, i.e.

- the backslash is removed, and
- whatever special function the character following may have ...
 - ... **it is disabled**
- (so if it has none it simply stands as it is).

Note that the backslash may also be used to quote itself, i.e.:
\\ (**when** quoting is considered) →
\ (**after** quoting has taken effect).

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



Quoting with Double Quotes

Until the next **unquoted** double quote^{*}

- **no** newline or semicolon will be taken as **command separator**,
- **no** horizontal white space 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 Examples](#).

^{*}: Double quotes are what in German is known as *Gänsefüßchen*.



Quoting with Curly Braces

It this way

- every character inside a range enclosed in curly braces is disabled with respect to its special function in [Tcl's Syntax Analysis](#),
- except that contained curly braces are **counted**
 - to find the *final* closing brace,
 - matching the *initial* opening brace, and
- back-slashes **do not change** any content inside the curly braces,
 - but may be used to **disable counting**,
 - if it directly prefixes an (opening or closing) curly brace.

Especially be aware that backslashes **will not be removed** from a range enclosed in (matching) curly braces!

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



Quoting 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 `-nonewline` 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 Examples (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 from the speaker – 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: The Programming Language Tcl

- Understanding the Syntax (quickly*)
 - 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 by the speaker.

*: This is a *very short and condensed* recapitulation of the section on the [Tcl Syntax](#), for all who missed attending to [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 it is 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-) key-based look-up tables,
 - (aka. Hashes in some other scripting languages)
 - (so rather like `std::map<string, string>` in C++).

Recently Tcl has added another important data structure:

- **Dictionaries***
 - structurally similar to Arrays, with
 - nesting over multiple levels (with – in principal – unlimited depth).

*: Which also have quite a different access style, borrowing some of its concepts from Functional Programming. Therefore this topic potentially can get large, but as Tcl dictionaries are not used in Vivado, they will receive no further coverage in this presentation.



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 }
```

*: Read carefully "*applied to lists*" – i.e. the **content** – **not** "*applied to list variables*" – the **name**. Therefore `month_names` is always prefixed with a `$` above!



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 "$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 suggests ...

```
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 as long as neither the content of the list nor the value appended is predictable and may hold problematic content. (E.g. assume `other` holds any 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(February))
```



Arrays 101 - Nested Indices and Multiple Dimensions

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

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

... that way:

```
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.

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

But ambiguity here!

```
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) ...

If this is given as presentation ...

... feel free to propose (more) things to try out.

*: 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?



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 concatenates its arguments to a single, long string, which is then evaluated 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:

```
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 turns into a never ending loop for negative values.



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, with
- the final result that the Vivado command `opt_design` will **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` causes 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 a second **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 1st, 4th etc. arguments that hold the conditions (not the 2nd, 5th 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`.

For a list measurements

- holding
 - a location and
 - a temperature
- as two-element sub-lists,

the following example generates a detailed report with an average of all measured temperatures in the last line:

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



In Comparison: Counting Loops over Lists

Most Tcl developers consider foreach more elegant and better readable in comparison to

- a classical counting loop over
- the **number of** list elements

accessing individual elements **inside** the loop via their index:

```
set count [length $measurements]
set sum 0.0
for {set i 0} {$i < $count} {incr i} {
    lassign [lindex $measurement $i] loc temp
    # or even more verbose:
    # set loc [lindex $measurement $i 0]
    # set temp [lindex $measurement $i 1]
    puts "temperature at $loc is $temp"
    set sum [expr {$sum + $temp}]
}
puts "average temperature is [expr {$sum / $count}]"
```



Variation: Loops with foreach - Parallel List Traversal

A different form of `foreach` can be useful in a similar example, which assumes

- locations and
- temperatures

stored in two **separate** 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.



Variation: 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

- temperatures stored in an array `measurements`,
- indexed by their location

(same for both fragments):*

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

Or with yet an alternate loop form:

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

*: No code to calculate the average is provided here but it should be obvious how to add it.



Early Loop Termination or Re-Evaluation Loop Continuation

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 the result is left as Tcl-list in primes when 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 ask_how_many {} {  
    global N  
    puts -nonewline "how many primes? "  
    flush stdout; gets stdin N  
}  
proc calculate_primes {} {  
    global N primes  
    ... ;# as before  
}  
...  
ask_how_many  
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.

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



Communicating through Reference Arguments (1)

Yet another communication to return results are *reference arguments*, requiring

- on the *caller's side* to hand over a variable **name**, and
- on the *callee's side* to link that name via **upvar** with a local alias.

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

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



Communicating through Reference Arguments (2)

To properly use `upvar` it is essential to understand that via with this command a subroutine "reaches out" to variable "belonging" to its caller.

Below the technique is extended to `calculate_primes` which now

- returns the **list of primes** also via a reference argument

(and an empty string as return value):*

```
proc ask_how_many {vcnt} {  
    ... ;# as before  
}  
proc calculate_primes {N vprimes} {  
    upvar $vprimes primes  
    return "" ;# (otherwise would return result of last command)  
}  
# handing over variable(s) for read / for modify  
ask_how_many N ;# (none) / `N`  
calculate_primes $N primes ;# `N` / `primes`  
puts "first $N primes: $primes" ;# `N` and `primes` / (none)
```

*: Many Tcl programmers allow themselves some sloppiness with respect to return values of subroutines, that way saving the effort to type explicit return statement ... but potentially making their code a bit less robust, as now a subroutine might return the "right" value just by chance.



Arguments with Default Values

The following example once more expects its first argument to be passed by reference (again applying upvar in cookbook-style) and also demonstrates the use of a default value for 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)
```

*: Note that fincr does not fully mirror the more recently modified behavior of incr to initialise the variable handed over as first argument with zero if it is not yet defined.



Variable Length Argument Lists

Variable length arguments lists have already been used ... in the helper function used a number of times to show the resulting command line after Tcl has finished its syntax analysis.

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 String

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` - three-way compare (**== 0 means equal !!**)
- `string equal` - compare exactly for equality (**!= 0 means equal**)
- `string match pattern ...` - compare with *pattern* (according to file name pattern matching rules)

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

^{*}: Be aware of character set issues with this and all other string handling commands. While portably it is safe to assume – since long – that internally Tcl uses at least 16 bit characters (i.e. the Unicode **Basic Multilingual Plane (BMP)** is completely covered), support for the U+010000 to U+0FFFFFF range may still have some flaws.



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 standard 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.*



Named Character Classes

The portability problems with international character sets and character ranges defined by code points is well known since long.

As a solution **Named Character Classes** are provided ...

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

... with **Single Letter Escapes**:^{*}

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

This also helps with internationalization, as if necessary definitions like

- `[:alpha:]`,
- `[:upper:]`,
- `[:lower:]`,
- ... (etc.) ...

can be switched according to the locale chosen.

^{*}: Such escapes are – of course – only recognized and evaluated as part of regular expressions.



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:**^{*}
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 + or -
\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.



Regular Expression Advanced Examples

As an advanced example consider a regular expression describing the currency formatting as usual in Germany:

```
set CURRENCY_FORMAT {^\d{1,3}([\.]\\d{3})*,\\d{2}$}
# from the start-----^| | | | | | | | | | | | | | | |
# one to three digits-^^^^^^^| | | | | | | | | | | |
# more groups of three digits-^^^^^^^| | | | | | | | | |
# repeated any number of times -----^| | | | | | | |
# followed by comma and two more digits-^^^^^^^|
# and up to the end nothing after it-----^
```

As some easy variations consider an optional decimal part (1) or a mandatory decimal part that may be replaced by a dash (2):

```
set CURRENCY_FORMAT {^\d{1,3}([\.]\d{3})*(\,\d{2}$)?} ;# 1
set CURRENCY_FORMAT {^\d{1,3}([\.]\d{3})*,(\d{2}|-)} ;# 2
```



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 -- $value {  
    {^X"[[:xdigit:]]+"$} {  
        ... ;  
    }  
    {^[01]+"$} {  
        ... ;  
    }  
}
```

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

*: The character classes specified as range 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.

- `clock seconds` – return current time in seconds resolution
- `clock milliseconds` – as before but milliseconds resolution
- `clock microseconds` – as before but microseconds resolution
- `clock format ...` – generate string from internal representation (with formatting details supplied in further argument)
- `clock scan ...` – generate internal representation from string
- `clock add ...` – helper for time/calendar calculations

Determine same time yesterday:

```
set t [clock add [clock seconds] -1 day]
```

Return current time as string (YYYY-MM-DD hh:mm:ss):

```
proc time_stamp {} {  
    set now [clock seconds]  
    clock format $now -format "%Y-%m-%d %H:%M:%S"  
    # implicitly returns result from last command  
}
```



Working with Files

Working with files basically falls into three main categories:

- Working with Directories
- Operations on Entire Files
- 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;*
- `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 **per-process resource**, be careful when changing the current working directory, as it determines how relative file path names are resolved for all file system accesses of the program.

*: 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 directory to ...
...          ;# do work in ...
cd $old      ;# restore previous 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 U*ix traditionally these are all files that have a name beginning with dot.



Operations on Entire Files

The command `file` has a large number of sub-commands for all many operations that apply to file in its entirety (not the content of a file).

Check for existence and given properties:

- `file exists ...` - check whether some a file exists
- `file readable ...` - check if it can be opened for reading
- `file mtime ...` - return modification time
- `file size ...` - return size of file
- `file isfile ...` - check for regular file
- `file isdirectory ...` - check for directory
- ...
- `file type ...` - determine type
- `file stat ...` - returns **all** properties in Tcl array

Print all properties of current working directory:

```
file stat [pwd]
foreach {property value} [array get stat] {
    puts "$property: $value"
}
```



Operations on Whole Files (cont.)

Operations with file path names:

- `file tail ...` - return last component
- `file rootname ...` - return last component without suffix
- `file extension ...` - return suffix of last component
- `file dirname ...` - return everything except last component
- ...
- `file separator` - get OS-specific path separator
- `file join` - concatenate arguments to pathname
- `file split ...` - separate components into list of parts
- ...
- `file normalize` - replace contained parts, e.g. `..././...` or `..././...`
- `file nativename` - turn into OS-specific form
- ...

Change to directory one level up from home directory:

```
set home [file normalize ~]
cd [file dirname $home]
```

Replace current suffix of file `$f` with `.bak`:

```
... [file join [file dirname $f]\
      [file rootname $f] .bak] ...
```



Operations on Whole Files (cont.)

Create Directories:

- `file mkdir ...` create directory (one or more)

Copy and Move Files:

- `file copy` - file to file (or directory)
- `file rename` - file to file (or directory)
- `file copy dir` - files to directory
- `file rename dir` - files to directory
- `file copy -force ...` - silently overwrite (variants as before)
- `file rename -force ...` - silently overwrite (variants as before)

Create backup copy of file with suffix changed to bak:

```
proc backup {file {bak .bak}} {  
    set dir [file dirname $file]  
    set base [file rootname $file]  
    set bakfile [file join $dir $stem $bak]  
    file copy -force $file $bakfile  
}
```



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 put),
 - 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 handling to the caller of a function.

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

```
proc get_file_lines {name} {  
    set fd [open $name]; set rdata [read -nonewline $fd]; close $fd  
    split $rdata \n; # result of `split` implicitly returned here  
}  
...  
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 there is a risk the above might succeed on open but fail on read, there is a resource leak, as the 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, ...),
- various translation modes, e.g.
 - no translation at all (binary mode), or
 - CR+LF (\r\n) → LF (\n only) on input and LF (\n only) → CR+LF (\r\n) on output,
 - ...
- whether read will block until data is available.*

Non-blocking reading is often used for "pseudo files" representing serial lines (UARTs), TCP/IP-sockets, or U*ix pipelines.

In some cases it may also make sense for console input or regular files.

*: Be sure to understand that the use of non-blocking I/O may lead to wasted CPU-cycles and/or sub-optimal latency if the program logic uses "busy waiting". Asynchronous designs with read operations in call-backs (registered via `fileevent`) are usually the superior **though not easily available with the Tcl-integration in Vivado**.



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** for `fconfigure` but in section [Serial Communications](#) of `open`.

The example below shows how to open and configure a UART device file:*

```
set fh [open /dev/ttyS0 rw]
fconfigure $fh -mode 19200,n,8,1 -blocking none -buffering none
fileevent $fh readable rec_byte; fileevent $fh writable snd_byte
```

*: Assuming the device is named `/dev/ttyS0`, transmission parameters are **19200 baud, no parity, 8 data bits, 1 stop bit, non-blocking I/O with no buffering**, and callbacks are to be registered for being called whenever a data-byte is ready available (`rec_byte`) to be read or can be send (`snd_byte`).



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`):

```
# fconfigure $fh -blocking read
# ^^^^^^^^^----- only necessary if not the default
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 none
# ^^^^^^----- necessary as usually NOT the default
while {![eof $fh]} {
    set ch [read $fh 1] ;
    #      ^^^----- will NOT block when no data available
    if {[string length ch] == 0} {
        # ^^^^^^^^^^^^^----- so it is necessary to check success ...
        ... ;# ... and something really useful should be done here
    } else {
        ... ;# (unless the received character can be processed here)
    }
}
```



Really useful means exactly that: **REALLY USEFUL** – doing nothing (and simply repeat) would eat-up precious CPU time!*

*: 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. Though, this requires a trade-off between lowering the CPU load and increasing 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 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.
- With **only** a file-handle as argument it returns the next line read,

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.

Principles of both usage forms (assuming `fh` refers to an open file):

```
... [gets $fh ln] ...    ;# reads line into `ln` (strips away NL and
                        # returns line length or `-1` for EOF)
... [gets $fh] ...      ;# returns line read (with NL stripped away,
                        # EOF could be checked for with `feof $fh`)
```



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`).

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 fh [open $name w]  
    puts $fh [join $content \n]  
    close $fh  
}
```

^{*}: Why did the example **reading** lines into a long string and then splitting its content into a list **had** to use `-newline` with the `gets`-command, but when **writing** lines generated from a list using `-newline` with the `puts`-command were wrong?



More Operations with Open Files

The commands `eof`, `seek`, `tell`, support more operations with files (via file handles typically obtained from `open`).

Printing the last ten bytes of a (presumably large) file in hex:

```
set fh [open ~/some_large_file rb]
seek $fh -10 end
set offset [tell $fh]
set chunk [read $fh]
foreach byte [split $chunk ""] {
    scan $byte %c byteval
    puts [format "byte %d: %02x" $offset $byteval]
    incr offset
}
if {[eof $fh]} {
    puts "<EOF>"
}
close $fh
```

*: Just to demonstrate the use of Tcl's `eof`-command, at the end the example above makes sure it actually has reached end-of-file. This should of course always be true, as first `seek` goes 10 bytes back from the end and then `read` - without third argument - reads everything from there up to the end. When reading **fewer** bytes the output will not end with the `<EOF>`-message. Interestingly, it will also not end with this message if **exactly 10 bytes** are read. What does this tell about EOF detection? (If you grew into file processing by using C on Unix you will probably not be surprised.)



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.^{*}

^{*}: If this is critical you will usually have to add provisions on at least two more levels:
(1) tell the operating system **not** to buffer any file output to this file or file system and
(2) disable potential chaching by (disk) drives.
Also understand that it might be very hard to **reliably demonstrate** this is effective.



Command Line and 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 to 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`).



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  
    # ^^^^^^^^^^^----- actually not necessary, because globals  
    #                vv----- can alternatively be prefixed with `::`  
    puts stderr "$::argv0 [FATAL]: $message"  
    exit $exitstatus  
}
```



A call to `exit` will usually terminate a Tcl script immediately, 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 often chose to ignore `exit` or at least do all necessary cleanup and may also save any data important 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 Foreground Processes

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] ;# `exec` as child process, wait for end,
# ^^^^^^----- and receive any output via return value
```

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



Running Background Processes

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 &      ;# `exec` as child process, do NOT wait
                  # ^----- (requested ampersand at end of command)
```

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 &] ;# `exec` child process, do NOT wait
# ^^----- receive PID (or list of PIDs for pipe)
```

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 Background Processes

Saved PID(s) may be used to control the background process in some way. Most often this is either to ...

... test whether it is still running ...

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

... or to forcefully terminate it.

```
# 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>



Receiving Output from a Process

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

A pipeline to a child process is created by the `open` command when its "file name" argument starts with a vertical bar (`|`).

Therefore the usage in principle looks as follows:

```
set xprog "... .." ;# program to execute (with arguments)
set fh [open |$xprog r] ;# run as child process with a pipeline
#  ^^-----^-----^----- returning a file handle to READ from
```

The file handle can be used afterwards for reading, as further outlined on the next pages,

- either line by line – typically in case of textual data,
- or in arbitrary chunks – useful for unstructured or binary data.

To avoid resource leaks, **do not forget this** at the end: `close $fh`



Receiving Output from a Process (cont.)

Typical fragments may look like one of the following two, processing the data sent by the child process either line by line ...

```
while {[gets $fh line] >= 0} {  
    ...  
    ... ;/# process single lines as it is received  
    ...  
}
```

... or in arbitrary chunks as it becomes available:

```
fconfigure $fh -translation binary  
while {![eof $fh]} {  
    set data [read $fh]  
    set n [string length $data]  
    ...  
    ... ;/# process chunk of `$n` bytes as it is received  
    ...  
}
```



Supplying Input to a Process

The `open` command may also be used to start a child process to which it supplies standard input in data chunks, produced piecemeal.

First the main part that

- starts the child process connected via a pipeline, then
- continues with a loop calling the function to produce data

(as was shown on the next page):

```
set xprog "... .." ;# program to execute (with arguments)
set fh [open |$xprog w] ;# run as child process with a pipeline
#  ^^-----^-----^----- returning a file handle to WRITE to
while {[produce_more data]} {
    puts $fh $data
}
...
close $fh ;# <-- tell the child process no more data will follow
```



Supplying Input to a Process (cont.)

Now the helper function to produce a single data chunk that subsequently will be sent:

```
proc produce_more {vdata} {  
    if { ... } { ;/# more data input to the child process  
        upvar $vdata data  
        set data ... ;/# whatever more data is available  
        return 1  
    }  
    return 0 ;/# data for child process exhausted  
}
```

The function is designed to be called repeatedly until it indicates the data is exhausted by returning the value 0.

Note that this particular technique is especially attractive as it does **not** require all data is readily available when the child process starts.



Potential Problems with Pipeline Architectures

Especially when the ability to **produce** the data to be sent **depends** on the progress of the child process **consuming** that data.

So beware of possible dead-locking scenarios!*

Some possible precautions are:

- Flush buffers if a reaction on the data sent is expected.
- Maybe avoid blocking reads,
 - or use time-outs instead of unlimited waiting ...
 - ... and try to find to a trade-off between
 - busy waiting and
 - sub-optimal latency
- **or even better strive for a fully event-driven architecture.**

*: Sometimes the root cause for such problems is data buffering you are not aware of. It may be done in higher levels (like the I/O channels of the Tcl shell), libraries used (like the C standard library file handling layer) or in the pipeline mechanism (as implemented by the operating system).



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`.^{*}

^{*}: 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).

The speaker will show live examples 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 "").

The speaker will show live examples if this topic is of special interest.



Part 4: The Tcl-Vivado Integration

- Accessing the Vivado Design Suite
 - Using the Vivado GUI
 - Using Vivado Tcl Commands
 - Accessing the Tool Command Language
 - The Vivado Tcl-Console
 - Suspending the Vivado GUI
 - Vivado Batch Script Mode
-



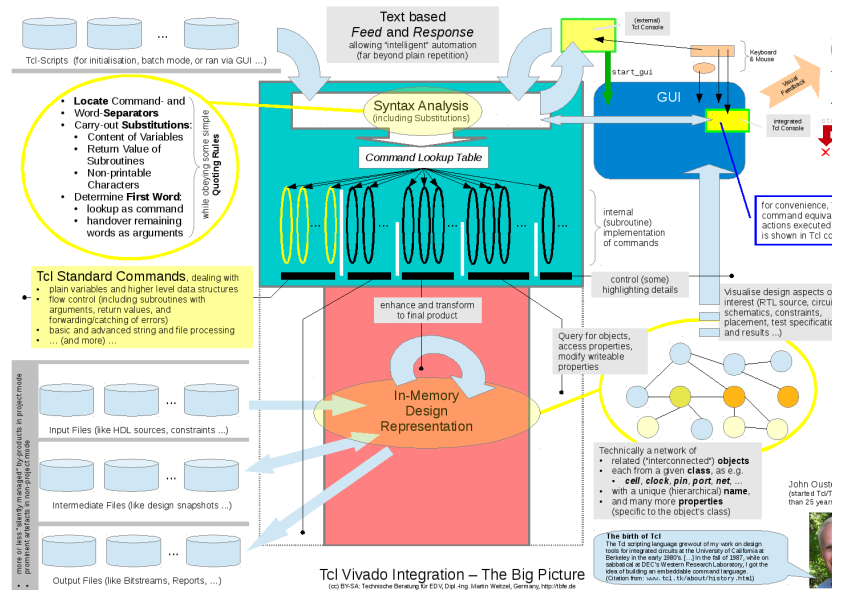
Accessing the Vivado Design Suite

There are two major ways to work with Vivado:

- Via GUI
- Via Tcl Commands

The GUI is usually brought up first when Vivado is started

- by clicking on its icon or
- by specifying the path to its executable on the command line.*



*: At the command line the option `-mode tcl` or `-mode batch` may be used which would not show the GUI immediately but only when requested.



Using the Vivado GUI

For many first-time users Vivado may appear to be a GUI based tool, accessed in well-known ways:

- Present *State Information* in various "Graphical Views", which the user can open and close, resize and arrange on the screen.
- *Initiate Actions* by choosing from *Menus* or pressing *Buttons* provided in a Toolbars ...
- ... eventually prompting for action-specific details in *Pop-Up Dialogues*.

Participants of this training are assumed to be more or less familiar with using Vivado via the GUI.



Using Vivado Tcl Commands

Besides preparing and elaborating a Design Model via the Vivado GUI there is also a Programming Language Interface.

Using Vivado via that interface is at the center of this training.

Though in general everything that can be achieved via the Programming Language Interface can also be achieved via the GUI (and vice versa*), some deeper knowledge of the former will pay for

- automating systematic and repeated tasks,
- including decisions what to do next.

*: Though, some specific tasks may be easier in one of the two ways, and also both approaches may intersect by assigning a sequence of Tcl commands to a GUI menu or button.



Accessing the Tool Command Language

Technically most of the Vivado GUI is

- just an interface to an underlying data base – called the *Design Model*
- which can also be accessed – i.e. inspected and modified – via a *Command Language*.

More precisely, *Tcl* once developed by John Ousterhout in the mid 90s, has been chosen as *Tool Command Language* for Vivado.

Tcl is known for its

- terse,
- minimal,
- "shell"-like syntax ...*

... and providing core features like

- variables and data structures,
- sub-routines and flow-control,
- access to OS-services etc.

As a **Tool Command Language for Vivado** *Tcl* is extended with many new commands and application specific internal data types.

*: Compared more recent language developments Tcl's syntax may sometimes surprise rather by simplicity than by complexity but with its straight forward approach it lends itself very good for the job. The rich command set added by Vivado is recognizably modelled following a slightly different "style" than the native Tcl commands, which has its pro's and con's.



The Vivado Tcl Console

The Vivado GUI makes a [Tcl Console](#) available which is useful for

- Execute any Vivado Tcl Command
- but just [Watching the Tcl Console Output](#) is also helpful to **learn the equivalent Tcl command(s)** for actions invoked via the GUI, like
 - preparing the in-memory design model,
 - running synthesis or implementation,
 - generating reports, or
 - for viewing or
 - modifying design objects
 - as part of design elaboration,
 - writing bitstream-files, or just

Watching the command equivalent for GUI actions in the Tcl console is an extremely useful approach* if you want to find out which commands are necessary to automate a systematic task.

*: Besides [Extracting Tcl Commands from a Journal](#) as is discussed later.



Vivado Tcl Console Details

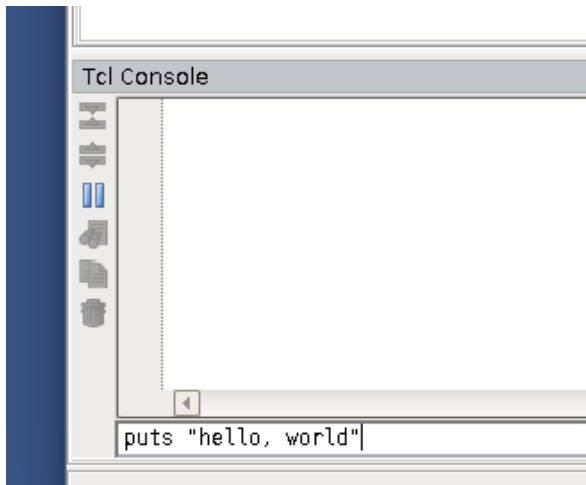
The Vivado Tcl Console consists of two main parts:

- An **Output Window**, showing
 - commands executed together with
 - output generated (on success) or
 - error messages (on failure)
- A **Command Line**, to
 - enter commands via the keyboard,
 - supporting
 - command history
 - command completion
 - command editing
 - and syntax highlighting



Vivado Tcl Console Command Line

The Tcl Console Command Line is located below the Output Window.



The command line can be edited while typing with the usual means:

- Try Cursor-Left and -Right, Backspace, Delete etc.

It also makes previous commands available via its history feature:

- Try Cursor-Up and -Down.

Finally command completions are proposed from which one may be selected:*

- Try TAB while a command typed-in is not yet complete.

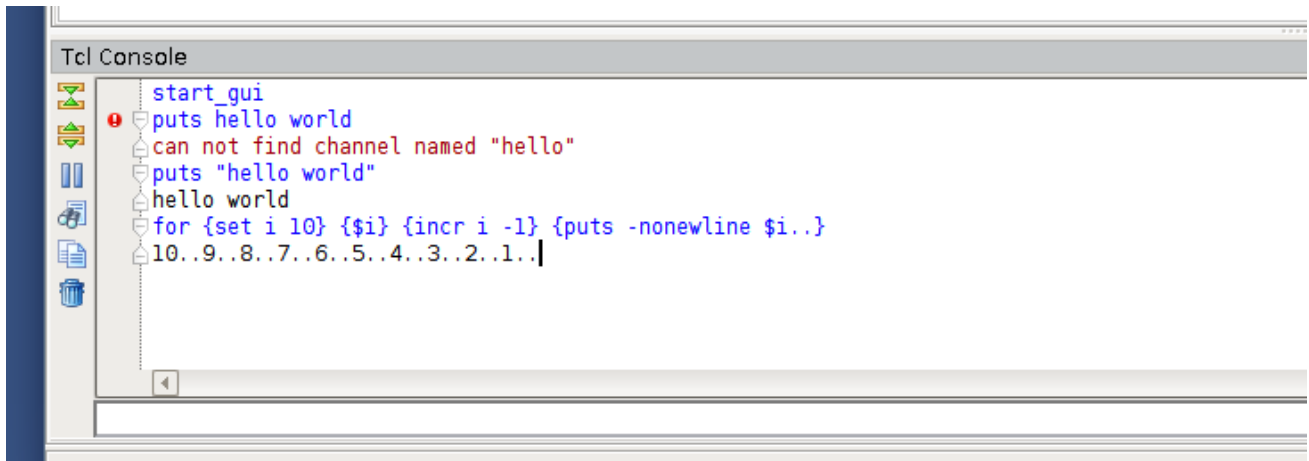
*: As completions are as soon as the window loses focus, it can not be captured with the screen-grabbing utility.



Vivado Tcl Console Output Window

Output of Tcl Commands is shown in the window **directly above** the command line, displaying

- executed commands in blue, followed by
- ordinary output in black, or
- error messages in red (preceded by an exclamation mark).



The screenshot shows the 'Tcl Console' window in Vivado. The command line on the left contains the following commands: `start_gui`, `puts hello world`, `can not find channel named "hello"`, `puts "hello world"`, `hello world`, and a `for` loop. The output on the right shows the results of these commands: `start_gui` (blue), `puts hello world` (blue), `can not find channel named "hello"` (red, preceded by an exclamation mark), `puts "hello world"` (blue), `hello world` (black), and the output of the `for` loop (black).

```
Tcl Console
start_gui
puts hello world
can not find channel named "hello"
puts "hello world"
hello world
for {set i 10} {$i} {incr i -1} {puts -nonewline $i..}
10..9..8..7..6..5..4..3..2..1..|
```

Some more features may be clued from the tool buttons and decorators.*

*: Take some time and try more things yourself, e.g. clearing all and folding all or parts of the output.



Suspending the Vivado GUI

After starting the Vivado GUI – which is the default when running Vivado by clicking on its icon or from terminal (= shell command line) ...

... entering the Tcl command

```
stop_gui
```

will temporarily suspend the GUI and place Vivado in *Tcl Mode* ...

... i.e. now you work with a raw *Tcl Shell*, from which the command

```
start_gui
```

returns to the Vivado GUI.*

In *Tcl Mode* you have all the commands available that you may use at the command line of the *Tcl Console* in the Vivado GUI.

In most cases you will probably prefer the Vivado GUI, as it also has command history, editing, completion and colorizing.

*: If you have looked closely to the example showing an excerpt of what was displayed as [Tcl Console Output](#) you may have noticed a `start_gui` command already there. It is kind of an artifact from Vivado Starting-Up, finally switching GUI mode (unless started in `-mode batch -mode tcl`).



Vivado Batch Script Mode

If the only thing to run in a Vivado session is a script with (prepared) Tcl commands, another option is to use *Tcl Batch Script Mode*.

The following will just run `my_script.tcl`:^{*}

```
vivado -mode batch -source my_script.tcl
```

It is equivalent to running Vivado in in *Tcl Mode* with

```
vivado -mode tcl
```

and then enter the command

```
source my_script.tcl
```

or to type this to the command line of the Tcl console command after starting up the Vivado GUI.

^{*}: Assumed here is that Vivado (i.e. the executable program name) is in your command search path so that the command `vivado` runs the design suite.



Part 5: Using Tcl in Vivado

- The Big Picture
 - Zooming in to ...
 - Tcl-Lists and List Handling
 - Vivado Design Object Access
 - Vivado Design Object Naming
 - Vivado Design Object Relations
 - Vivado Result List Filtering
-



The Big Picture

Under the surface of Vivado there are ...

(... many things ...)

represented by the internal

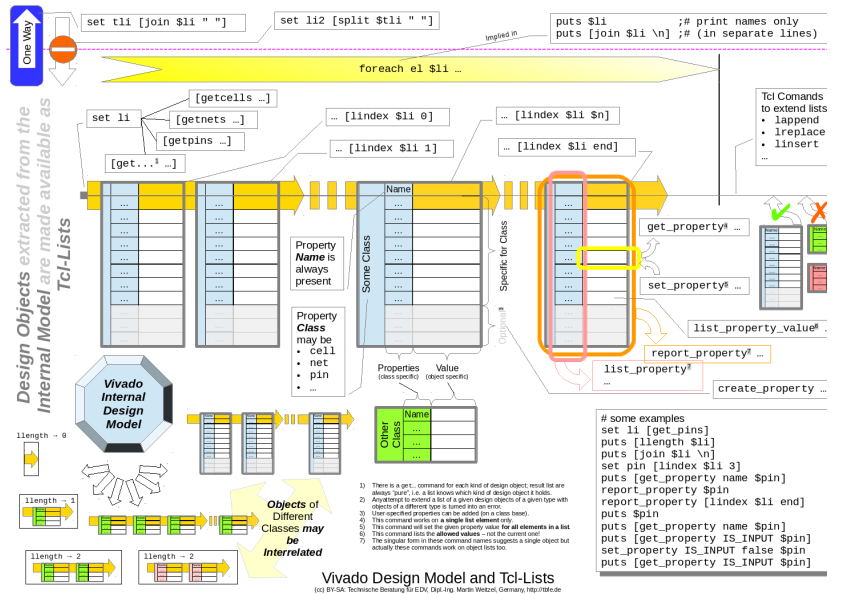
- Design Model

... but here the focus is on:

- *Design Objects*

and their

- *Relations.*



Zooming in to ...

What you need to now in Detail:

- Some Basics on Tcl-list Handling
- How to access a design object held in a Tcl-list
 - to read its attributes
 - or modify its attributes
- How you reference a design object by
 - its name (which may be hierarchical)
 - relations to other design objects



Tcl-List Handling

To explain the vrey basics of Tcl-list handling, let's step away from Vivado for a moment.

- Take this example:

```
set li "Berlin Hannover Stuttgart Nürnberg"
```

Does `li` hold a (Tcl^{*}-) list now?

Yes and no, actually one can argue that `li` holds just a string but ...

- ... let's see:
- Or try this:

```
llength $li      ;# -> 4
lindex $li 0     ;# -> Berlin
lindex $li 1     ;# -> Hannover
lindex $li end   ;# -> Nürnberg
lappend li "München"
llength $li      ;# -> 5
```

```
puts "--> $li <--"
puts [join $li \n]
puts [join [lsort $li] \n]
puts [lrange $li 0 1]
puts [lrange $li 0 100]
puts [lrange $li end-2 end]
```

^{*}: As we are focussing on Tcl-lists in this section, please read the un-prefixed word "list" as "Tcl list".



Creating Tcl-Lists

The first thing to note is this:

You may apply list-related commands to plain strings* ...

... but there are also Tcl commands that return their result as a list:

```
set files [glob *.log]    ;# LIST of *.log-files in working directory
puts [join $files \n]    ;# printed one name per line
foreach f $files {puts $f} ;# (as before with explicit loop)
foreach f $files {
    puts "=== $f ==="
    file info $f attrs    ;# array `attrs` holds attributes of `$f` ...
    foreach {name value} [array name attrs] { ;# turned into LIST ...
        puts "$name: $value"    ;# (of name-value pairs) and printed
    }
}
```

*: There are certain limitations to the string being "well-formed" if Tcl's list-related commands are to be applied. E.g. the string "Berlin Hannover {Frankfurt am Main} Stuttgart" is also a well-formed list, as is "Berlin Hannover Frankfurt\ am\ Main Stuttgart".

But "Berlin Hannover {Frankfurt am Main Stuttgart" is not!

Finally, "Berlin Hannover Frankfurt am Main Stuttgart" again defines a string that is also a well-formed list, but – when list-related commands are applied – with a different content as the first ones.



Design Object Lists

Vivado has added a number of specific commands to Tcl to create

Lists of Design Objects

What actually makes a Design Object is covered in a minute or two, first one example for such a command:

```
set li1 [get_files]           ;# list of files HDL in current project
set li2 [get_files *]         ;# as before (`*` is just the default)
set li3 [get_files *.vhdl]    ;# limited to files with suffix `.vhdl`
```

Accessing objects in the Vivado Design Model will be covered later in much greater detail, but to give a first idea:

- get_cells → list of cells
- get_nets → list of nets
- get_pins → list of pins
- ... (etc. etc.) ...

So far this should only illustrate
the **General Concept**
how to retrieve
Lists of Design Objects from
Vivado's Internal Design Model



Turning Lists Into Strings ... and Back

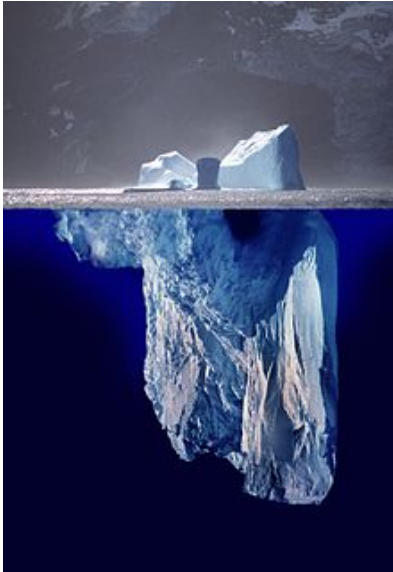
Like some Tcl commands generate a result as list from a string, others work vice versa, **creating strings from lists**:

```
set li {Mainz Wiesbaden Frankfurt\ am\ Main}
puts $li                                ;# show "as is" (remember output)
puts [llength $li]                     ;# -> 3 (plausible)
set cities [join $li \n]                ;# STRING from list
puts $cities                            ;#
puts [llength $cities]                  ;# -> 5 (why that?)
set city_list [split $cities \n]         ;# LIST from string
puts [llength $city_list]               ;# -> 3 (OK again)
puts $city_list                         ;# (compare to remembered output)
```

But what is the importance of this for lists of *Design Objects*?

Design Objects are like Icebergs ...





Above the surface only the tip of the iceberg is visible ...^{*}

Design Objects can be viewed as **Units of Information**.

The detailed content, i.e.

- the set of *Properties* and
- the respective *Values*.

depends on their **Class**.

- A common property of all classes is the **Name**.

A design object is **reduced to its name only** whenever it is used in a context where Tcl expects a string.

^{*}: For an Iceberg it's about 15% of its mass (or volume), for a *Design Object* it's just its name.



Using Design Objects as Strings

The fact that any design object can be used in a string context is mostly useful to simply print the names of objects retrieved with some of Vivado's respective `get_`-commands:

```
set li1 [get_cells] ;# retrieve all cells (of a synthesized design)
puts [llength $li1] ;# maybe a bit large ... ?
set li2 [lreplace $li1 3 end] ;# ... so keep just the first three
puts [llength $li2] ;# three now - ok,
puts $li2 ;# show them (or rather their names)
```

So far – so good – (and so useful) – **BUT** ...

... now you are dropping all the payload, keeping only the name:*

```
set li3 [join $li2 \n]; puts $li3 ;# all 3 names, one per line
set li4 [split $li3 \n]; puts $li4 ;# still looks good, doesn't it?
```

Just wait until you know how to access a *Design Object's* properties!

*: In practice this is a minor glitch not happening that often, but it is crucial for a solid understanding of the true nature of *Design Objects*.



Once More, Printing Names

The names of all the design objects in list can also be done in an explicit (Tcl) loop and using the (Vivado) command to access the *NAME* property of the objects:*

```
proc print_names {obj_list} {  
    foreach obj $obj_list {  
        puts [get_property NAME $obj]  
    }  
}  
print_names $li1      ;# (maybe much output)  
print_names $li2      ;# (just the first three)  
print_names $li3      ;# OOPS! (maybe the contained '\n'-s?)  
print_names $li4      ;# Hmmmm? (obviously not)
```

*: As this is not a Tcl course some more Tcl features, especially for flow control get introduced "on the run". If this is a presentation given as talk on some conference or similar, feel to interrupt the speaker and ask for more detailed explanations, if the given ones are not sufficient to understand what's going on.



Object Lists vs. Single Objects

Note that `print_names` also works with single *Design Objects*:

```
print_names [lindex $li1 0]  
print_names [lindex $li1 end]
```

It seems attractive to create a modified version `print_one_name` taking an additional argument to select a specific item from the list item:*

```
proc print_one_name {obj idx} {  
    puts [get_property NAME $obj]  
}
```

Test it:

```
print_one_name $li1 0  
print_one_name $li1 end
```

*: If you **don't** think this is useful, be patient for a moment: this is just the prelude for more to come!



Merging print_names and print_one_name

This step deals mostly with Tcl programming – you may [skip it](#).

As an "on the run" example consider the next fragment:

```
proc print_names {obj_list {idx ""}} {  
    if {![string equal $idx ""]} {  
        set obj_list [lindex $obj_list $idx]  
    }  
    foreach obj $obj_list {  
        puts [get_property NAME $obj]  
    }  
}
```

Test it with the basics:

```
print_names $li2  
print_names $li2 0  
print_names $li2 end  
print_names $li2 end-1
```

And a bit pointless but anyway:*

```
for {set i 0} {$i<3} {incr i}\  
{  
    print_names $li2 $i  
}
```

*: The point is here to show a bit more of Tcl programming on the run to wet your appetite ...



Parametrizing The Property to Print

This step deals mostly with Tcl programming – you may [skip it](#).

One more extension to the "on the run" example seems natural now:*

```
proc print_property {prop obj_list {idx ""}} {  
    if {![string equal $idx ""]} {  
        set obj_list [lindex $obj_list $idx]  
    }  
    foreach obj $obj_list {  
        puts [get_property $prop $obj]  
    }  
}
```

Test it with the basics:

```
print_property NAME $li2  
print_property NAME $li2 0  
print_property NAME $li2 end  
print_property NAME $li2 end-1
```

And yet some more testing:

```
set cell [lindex $li1 end-1]  
print_property NAME $cell  
print_property CLASS $cell  
print_property CLASS [list]
```

*: Of course, it's another parameter so that not only the name but any other property can be printed.



Vivado Design Object Access

There are two basic categories

- *Design Object* Read Access (Retrieve Properties)
- *Design Object* Write Access (Modify Properties)

Note that not all properties are modifiable – some are also read-only.

Furthermore properties have a **type**, defining the valid set of values.

Numeric types:

- `bool` – 0 or 1
- `int` – 32 bit signed
- `long` – 64 bit signed
- `double` – floating point

Non-numeric types:

- `string` – any sequence of characters
- `enum` – a user defined set of symbolic names^{*}

^{*}: For the enum type the *Vivado Tcl Command* `list_property_value` returns a list of possible values in as symbolic names. (Internally Vivado probably implements these as integral numbers, though the mapping is not accessible.)



Accessing a design object that is already in a list usually means that:

The first step is to select a single list element.

(Of course, for a list with exactly one element this step does not apply.)

Then the following commands are relevant:

- `get_property`
 - retrieve a **single** property **value**
- `list_property`
 - retrieve some* or all property **names**
- `report_property`
 - retrieve some* or all property **names**
 - together with their **type** and
 - current **value**.

*: Which properties are shown can be limited via a (name) pattern. By default **all** properties are listed or reported. (Hence the singular form of the command name hints into the wrong direction – if it would end in ..._properties it were more appropriate for the default use case.)



Single Properties of Single Objects

The basic syntax is:

```
get_property [-quiet] [-verbose] <name> <object>
```



You can retrieve the full description from the *Vivado Tcl Shell* using `help get_property` or look it up in [XILINX UG835].

The above is copied and pasted from the *Vivado Tcl Shell* output.

It uses a typical meta-syntax which also works for non-"rich text" output:

- Most text is meant verbatim, **except** that
 - pairs of *Square Brackets* enclose optional parts;
 - an *Ellipsis*^{*} indicates optional repetition;
- and a *Word in Angle Brackets* represents
 - a brief symbolic identifier for its actual meaning.

^{*}: An ellipsis means three sequential dots and is not used as meta-syntax element in the example above.



Properties of Multiple Objects

Making an exception from the rule presented on the page before:

In certain cases `get_property` also makes sense for *Design Object Lists* holding more than one element.

The idea here is that

- you might **not** want to retrieve an *exact* property value,
 - which of course makes no sense if there are several design objects that can have different values
- but instead the *minimum* value or the *maximum* of all values.

On the last page the basic syntax (output of `help get_property`) was edited to hide the respective options. Here is un-edited* output:

```
get_property [-min] [-max] [-quiet] [-verbose] <name> <object>
```

*: At first glance it might surprise that the syntax seems to mandate just one single `<object>`, but actually this can be a list of *Design Objects* too.



Listing Properties

The basic syntax is:

```
list_property [-class <arg>] [-regex] [-quiet] [-verbose]  
               [<object>] [<pattern>]
```

The optional <pattern> is a way to limit the output to only those properties that match that pattern.



You can retrieve the full description from the *Vivado Tcl Shell* using `help list_property` or look it up in [XILINX UG835].

Clearly that command refers to classes, not individual objects, therefore:

- **either** the class may be specified after `-class`
 - then the <object> argument must be omitted;
- **or** an <object>^{*} is specified and used to retrieve the class.

^{*}: Note that lists of more than one Design Object are never problem, as it is impossible to creates such lists with mixed classes.



Reporting Properties

The basic syntax is:*

```
report_property [-all] [-class <arg>] [-return_string]
                [-file <arg>] [-append] [-regexp] [-quiet]
                [-verbose] [<object>] [<pattern>]
```



You can retrieve the full description from the *Vivado Tcl Shell* using `help report_property` or look it up in [XILINX UG835].

In case you are reading this text silently yourself and have a Vivado installation at hand, you are now encouraged to practically try the commands for **retrieving** information held in *Design Objects*.

If this is given as presentation ...

... feel free to propose (more) things to try out.

*: Again the optional <pattern> limits the output to only those properties matching that pattern.



Vivado Design Object Write-Access

It this case there is just one command to learn:*

```
set_property [-dict <args>] [-quiet] [-verbose] <name>  
             <value> <objects>...
```



You can retrieve the full description from the *Vivado Tcl Shell* using `help set_property` or look it up in [XILINX UG835].

In case you are reading this text silently yourself and have a Vivado installation at hand, you are now encouraged to practically try the commands for **modifying** information held in *Design Objects*.

If this is given as presentation ...

... feel free to propose (more) things to try out.

*: There is another command, `reset_property`, for which you may want to look the documentation and decide for yourself whether it might be useful for your project.



Vivado Design Object Naming

Until now the detailed coverage of one topic has been postponed:

- Which ways exist to create "*Design Object Lists*".

A typical technique is to either name one single object unambiguously or using a pattern to match a related group of objects.

- *Names are structured* with
 - an optional initial slash (/) and
 - eventually more separating slashes;*
- *Patterns for Design Object names*
 - employ the asterisk (*) as *Wild-Card Symbol*.

It will become obvious later that what is described here and on the pages following actually is a special case of [filtering](#).

*: The fact that Vivado even uses the same separator character as was introduced with Unix 40 years ago, might be taken as an indicator where the basic idea was drawn from. The separator can be changed with `set_hierarchy_separator`, though only to a limited set of special characters **not including the backslash** (as this probably would conflict with the traditional use of backslashes in Tcl's core syntax).



Understanding Hierarchical Names

Generally *Design Object Names* form a hierarchical tree-structure.

Hence a comparison with (usually well-known) *Navigation in Hierarchical File Systems* seems natural.

Design Object Names

- **if** starting with a slash
 - **then** are interpreted from the *Top of a Design*
 - **else** are interpreted from the *current_instance*
- followed by a (first)
 - name component.

File Path Names

- **if** starting with a slash
 - **then** are interpreted from the *Root Directory*
 - **else** are interpreted from the *Working Directory*
- followed by a (first)
 - name component.

Separated by more slashes, more name components may follow.
(In principle to any depth.)



Intermezzo: Block-Designs

What has been and will be explained for hierarchical names of *Design Objects* is also valid for *Block Designs*.

The only difference is that the commands are differently named:
Part of the *Block Designs* commands is a lexical `_bd_` infix.

(Ordinary) *Design Objects*

- `current_instance`
- `get_cells`
- `get_nets`
- `get_pins`
- `get_ports`
- ...

Block Design Objects

- `current_bd_instance`
- `get_bd_cells`
- `get_bd_nets`
- `get_bd_pins`
- `get_bd_ports`
- ...

If you want to try navigation via design object names on a simple example, consider to create a trivial *Block Design* as test case.*

*: Understand that the purpose of the following is NOT to have a "useful" block design but only to provide a simple set of components for trying navigation by (block) design object names: Create two sub-components, each holding a two- or three-bit *Value*, connect both with an *Adder* outside any of the sub-components, and put the whole thing into another component with pins for out-going connections.



Unambiguously Naming Design Objects

An unambiguous name may start at the top level of a design:

- `get_cells /aaa/bbb/ccc`
- `get_nets /aaa/uuu`
- `get_pins /aaa/bbb/zzz[3]`

Or start at a selected component:

- `current_instance /aaa`
- `get_cells bbb/ccc`
- `get_nets uuu`
- `get_pins bbb/zzz[3]`

Some more hints:

- Using the command `current_instance` without an argument (re-) sets the design top level as current instance.
- Like in file system navigation there is also the notation `".."` to refer to "the parent of `current_instance`".



For more information see [XILINX UG894]

→ *Accessing Design Objects* → *Getting Objects By Name*



Making Use of Wild-Cards

Wildcards in name patterns to denote *Design Objects* are straight forward.

Some examples:

- `/a*/bbb/ccc` – an object three levels below the design top level:
 - the first name component needs to start with an a;
 - the second name component needs to be exactly bbb;
 - the third name component needs to be exactly ccc.
- `a*/bbb/ccc` – as before but starting at `current_instance`.
- `/aaa/*b*` – an object two levels below the design top level:
 - the first name component needs to be exactly aaa;
 - the second name component can be anything with b in it.
- `x*y` – an object at the level below `current_instance`
 - with a name starting with x and ending with y



Empty result Lists

A result list* may be empty if

- a component specified with an unambiguous name does not exist;
- a pattern – probably intended to match – doesn't match anything.

Empty result lists are **not** an error but draw a warning.

- A result list (say stored in `li`) can be tested for empty or not with
 - `if {[llength $li] != 0} ...` – test for **not** empty
 - `if {[llength $li]}` ... – (as before)
 - `if {[llength $li] == 0} ...` – test for **empty**
 - `if {![llength $li]}` ... – (as before)
- (The warning can be silenced with the option `-quiet`.)

*: It may also result from further [filtering](#), with the same consequences as described here.



Applying the -hierarchical Option

As the previous examples have demonstrated, the number of components in a name is always fixed in the pattern, meaning:

Even with wild-cards a "search" for a component can only take place at a given "level" – relative to the design top level or `current_instance`.*

The `-hierarchical` option – often abbreviated as `-hier` – allows deal with this:

- `get_cells -hier abc` – at any level lookup cells named abc.
- `get_pins -hier x*y` – at any level lookup pins with a name
 - starting with x and
 - ending with y.

*: Note that a recursive search could be explicitly programmed by traversing the element hierarchy changing `current_instance` – as shown in [XILINX UG894] → *Accessing Design Objects* → *Getting Objects By Name* → *Using the -hierarchical option* (to illustrate the effect of this option).



Design Object Naming with Regular Expressions

Even more sophisticated search patterns for component names can be specified applying *Regular Expressions* – or REs in short.

As REs are not part of this presentation, the topic is given no further coverage.

In case you are already well-versed with REs, feel free to try some examples yourself.*

In case you are reading this text silently yourself and have a Vivado installation at hand, you are now encouraged to practically try the commands for **modifying** information held in *Design Objects*.

*: As a word of caution: Without being able to point to a specific case right now, the author of this representation once came to the conclusion that REs work sufficiently well but in some more advanced examples tried did not work according to the documentation, which may have one of three reasons:
(1) The Vivado documentation is correct but was wrongly interpreted.
(2) The Vivado documentation is ambiguous and allows different interpretations.
(3) The Vivado documentation differs from what is implemented.



Experimenting with and Patterns and the -hierarchical Option

It is strongly suggested that experiment and try unambiguous names and name patterns in a simple design, to solidify your understanding.

Just as an idea:

- Do not (only) try a number of things "interactively" and nod through the results (as long as they are or appear to be plausible).
- Instead – to a simple, fixed design – apply the TDD approach:
 - Write a short Tcl-script to generate various result lists.
 - Programmatically compare these lists to your expectation.
 - (So the comparison is also part of that script.)
 - **Of course, the final goal is to see "All Tests Passed".**

If this presentation is given as a talk and the time frame is not extremely narrow, the speaker will now give a live demonstration.

Feel free to propose (more) things to try out.



Vivado Design Object Relations

Another important way to navigate the *Internal Design Model* is this:

Follow the relationships certain *Design Objects* have with each other.

As an example:

- Some **cells** (may be retrieved by name with `get_cells`) are
 - connected to **nets** which
 - (may connect to (other) **cells**)
 - (and to other **nets**)
 - ... (etc. etc) ...
 - or terminate in **pins**.

This is expressed with the `-of_objects` option of the various commands producing result lists holding *Design Objects*.



For more information see: [XILINX UG835] → ... → *Getting Objects By Relationship* (as first overview), [XILINX UG894] → ... → *Object Relationships* (more details), and [XILINX UG912] → ... → *Netlist and Device Objects* (extended coverage).



Using the -of_objects Syntax

The syntax to trace *Design Object Relationships* looks like this:*

```
set li [get_pins -of [get_nets -of [get_cells /aaa/bbb]]]
```

Read: Get the pins connected to the nets connected to cell /aaa/bbb

Though the nesting of get_-commands is meant to (somewhat) mirror human speech syntax, it is often better readable to express long chains with intermediate results stored in (temporary) variables.

Once more the above example, accordingly re-written:

```
set start_cell [get_cells /aaa/bbb]
set nets_of_start_cell [get_nets -of $start_cell]
set li [get_pins -of $nets_of_start_cell]
```

*: Abbreviating -of_objects to -of is possible because the commands have no other option starting with -of **and is very common practice**. Because of the latter this will also be the case in future Vivado versions (though there is no expressed guarantee), as otherwise many existing scripts would break.



Example: Stepping (Visually) Through a Design

A nice way to get some practice

- navigating the *Internal Design Model*
- via *Design Object Relationships*

is to visually highlight the result, using `highlight_objects`.^{*}

^{*}: Here is a helper procedure ...

```
proc set_highlighted {key {objs {}}} {
    global highlighted
    if {[info exists highlighted($key)]\
        && [llength $highlighted($key)] > 0} {
        unhighlight_objects $highlighted($key)
    }
    if {[llength $objs] > 0} {
        highlight_objects $objs
    }
    set highlighted($key) $objs
}
```

... and some demo code using it – maybe a base for an "animated design walk".

```
set_highlighted my_cells [set c [get_cells]]
after 2000; set_highlighted my_cells [set c [lindex $c 0]]
after 2000; set_highlighted my_nets [set n [get_nets -of $c]]
after 500; set_highlighted my_cells; unset c
after 2000; set_highlighted my_nets [set n [lreplace $n 5 end]]
after 2000; set_highlighted my_nets; unset n
```



Vivado Result List Filtering

Result lists can also be limited by any condition (and combination of conditions) which can be expressible based on design object properties.

This is called *Result Lists Filtering* as the command option is `-filter`.

This sometimes causes problems (for developers with too little knowledge of the Tcl basics), especially with what is called "Quoting":

- White-space is significant in Tcl's core syntax as *Word Separator*
- The whole filter expression
 - is separated by white space from the option keyword `-filter`
 - and needs to be kept together as **one single word**,
 - i.e. **contained white-space must be quoted**.
- Therefore it is usually^{*} enclosed
 - either in *Curly Braces* – i.e.: `... -filter { ... }`
 - or in *Double Quotes* – i.e.: `... -filter " ... "`

^{*}: There are more ways to skin that particular cat, but these are the most common solutions.



Basic Filter Expression Syntax

Filter expressions have their own (sub-) syntax:

- As long as white-space is made invisible as *Word Separator* ...
 - ... filter expressions may contain any amount of white space*
- Otherwise the syntax loosely similar to C-style expression, with
 - referencing *Design Object Properties* is *case insensitive*, and
 - operators for comparisons based on pattern matching (=~, !~)

Filter expressions can be based on **any** *Design Object* property.

As a first example:

- `get_cells -filter {PRIMITIVE_LEVEL == LEAF}` - leaf cells at top
- `get_cells -hier -f {PRIMITIVE_LEVEL == LEAF}` - (at all levels)



For more information see: [XILINX UG894] → ... → *Filtering Results* and [XILINX UG835] → ... → *Filtering Based on Properties*

*: Which can much improve readability – if properly quoted even newline-s are acceptable.



Filtering on the *NAME* Property

Also possible is filtering on the *NAME* property, if it is to be used as selection criteria.

- Until now in this case it was always specified as ordinary argument.
- As a filter expression *NAME* comparisons are more powerful, as e.g.
 - they can be negated (not only but also for pattern matching) and
 - combined based on logic operations (&&, ||)

Some Examples:

```
get_cells -hier -f {NAME =~ */wr*reg} ;# like get_cells -hier wr*reg
get_cells -hier -f { !(NAME =~ */wr*reg) } ;# (negated match)
get_cells -hier -f { NAME !~ */wr*reg } ;# (negating operator)
get_cells -hier -f { NAME =~ */wr*reg || NAME =~ */rd*reg }

get_cells -hier -f { NAME =~ */wr*reg
                    || NAME =~ */rd*reg } ;# (using more white space)

get_cells -hier -f {NAME =~ */wr*reg || NAME =~ */rd*reg} ;# (less)
get_cells -hier -f {NAME=~*/wr*reg || NAME=~*/rd*reg} ;# (lesser)
get_cells -hier -f {NAME=~*/wr*reg||NAME=~*/rd*reg} ;# (even lesser)
get_cells -hier -f NAME=~*/wr*reg||NAME=~*/rd*reg ;# (no quoting)
```



Quoting Filter Expressions - Why and How?

Though compared to other languages the Tcl core syntax, it would break the narrow time frame for this talk.

Therefore – in extreme crash course style:

- White space needs to be quoted
 - if an argument is to be handled as "unit"
 - i.e. must not be broken into separate words.
- Quoting white space can be done with:
 - curly braces – `{... white-space doesn't separate words ...}`
 - double quotes – `"... white-space doesn't separate words ..."`
- The difference is between both is when it comes to substitutions:
 - `{... nothing inside gets ever touched in any way ...}`
 - `"... some standard substitutions will still happen ..."`

To summarize:

- Curly braces as quoting mechanism cause fewer headaches ...
- ... but the content of Tcl variables **is not substituted** inside.



Use Curly Braces
for handing over "plain untouched text".

Possible Pitfalls:

- If there are curly braces contained, they must be properly nested.
- It is impossible to have a backslash at the end.*

*: Backslashes are a problem if they are required because of their use in some VHDL identifiers which in turn affect the names used for certain *Design Objects* – see [XILINX UG835] → ... → *VHDL Extended Identifiers* for more details.



Use Double Quotes

for anything containing Tcl variables – say `my_var` – and for which the content is to be substituted (i.e. the actual use is `$my_var`).

Possible Pitfalls:

- Be aware of **all** the usual substitutions that may happen:
 - `$varname` gets substituted by the content of `varname`
 - `[some tcl command]` executes the embedded Tcl command

Carefully protect `$` and `[...]` if **not** meant to cause one of the substitutions above, by preceding each of it with a **single** backslash.

- OTOH, if the backslash does not **protect** one of the above, it **may** actually cause an unprintable character to be inserted.*

Diligently write **two** backslashes (`\\`) if you really need **one**.

*: Basically like in C/C++, i.e. newline for `\n`, tab for `\t`, etc.



Cookbook-Style Quoting: Recipe 3

In difficult cases, don't try to be overly clever, e.g. with nested quoting.

- E.g. it is very well possible to build an argument for the `-filter` option in advance, using a variable – say `my_condition`.
- There is a Tcl command `format` which is much like `sprintf` in C.
- You can use
 - curly brace quoting with contained place-holders and
 - "insert" required variable content via separate arguments.*
- Then the final use might look like this:
 - `get_cells -filter $my_condition`

*: To flesh-out the example a bit more, here is a fragment showing some details:

```
set my_condition [format {
  as the {format string} is curly-brace quoted not much surprises may happen,
  at least as long contained curly braces are properly nested, and when you
  need the content of variables - say `name` and `level`, do it this way:
    ... NAME ~= "%s" && PRIMITIVE_LEVEL == "%s" ...
} $name $level] ;#  ^^----- here and here -----^^ --- go the placeholders
# |||||  |||||  here go the actual variables for which the content is
# |||||  ^^^^^^ to be substituted (where the placeholders are used)
# ^^^^^^ \_____/
```



Quoting: A Word to the Wise

If you find yourself too often applying a
"Trial and Error Approach"
with respect to quoting:
Stop It!

- Tcl's core syntax is close to trivial.*
- It can be described with a tiny state machine.
- It takes not much time to learn and master it.

If you plan to do serious Tcl programming anyway (in a not too distant future), read a text book on Tcl or **consider to visit a Tcl training**.

- For the syntax basics and some useful library stuff **one day** suffices;
- Planning for little more – **say three days** – enables you to write small tools or Vivado extensions, and even start GUI Programming with Tk.

*: In fact, [John Ousterhout], the original designer of Tcl/Tk once stated. "*Many difficulties beginners may have with Tcl stem from the idea that Tcl's syntax might be more complex as it actually is.*"



User Defined Design Object Properties

The last feature covered in this talk is the extensibility of *Design Objects*:

User defined properties can be added with `create_property`.^{*}

E.g. you would add a user-specific boolean flag and a category accepting the three values Foo, Bar, and Baz as properties to pin objects this way:

```
create_property -type bool my_flag pin
create_property -type enum\
  -enum values {Foo Bar Baz} -default_value Foo\
  -description "My special category for `pins`"\
  my_category pin
```

Such properties are retrieved and modified as usual ...

- `get_property my_category`
- `set_property my_flag true`

... and can of course also be used in filter expressions:

- `... [get_pins -f my_flag] ...`
- `... -f {my_category==Foo} ...`

^{*}: For reasons unknown this is only supported for *Design Objects* of the following classes: design, net, cell, pin, port, Pblock, interface, and fileset



Example: Categorizing Objects Across Classes

The following code adds the new property `my_usage` to the classes `pin`, `cell`, and `net`:

```
set my_classes [list pin cell net]
foreach c $my_classes { create_property -type string my_usage $c }
```

Assuming `my_usage` has been set properly, the following helper returns all pins-s, cell-s, and net-s with a specific `my_usage` in an array:

```
proc get_some_objects {usage arrayname} {
    upvar $arrayname result
    unset -f result
    set condition [format {my_usage == "%s"} $usage]
    foreach c $::my_classes {
        set result($c) [get_${c}s -filter $condition]
    }
}
```

The intended use looks as follows:

```
get_some_objects "whatever" my_result
```



Example (cont): Categorizing Objects Across Classes

Finally a helper to apply any code fragment:

```
proc apply_to_all_of {arrayname code} {  
    foreach c $::my_classes {  
        list uplevel $code \${arrayname}($c)  
    }  
}
```

Its use to highlight in orange color all the *Design Objects* resulting from the call to `get_my_objects` (from the page before) in orange would look like this:

```
apply_to_all_of my_result {  
    highlight_objects -color orange  
}
```

To remove highlighting use this:

```
apply_to_all_of my_result {  
    unhighlight_objects  
}
```

