PLC2 - Workshop Tcl/Tk

Dipl.-Ing. Martin Weitzel
Technische Beratung für EDV
64380 Roßdorf
www.tbfe.de

Prelude
Discovering Tcl and Tk

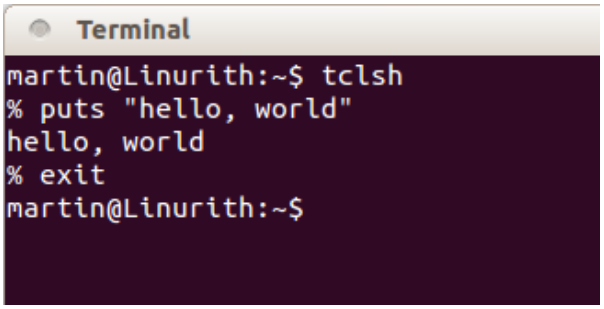
- What's ahead?
  - Using *tclsh* and *wish* interactively and to run scripts
  - A quick tour through Tcl's major features
  - The role of event driven designs
  - Simple Tk applications
  - Tcl as tool language
  - Interfacing with C/C++

Whenever you open this presentation later – after this course has finished – you are encouraged to open a command window on your computer and run a Tcl command interpreter (`tclsh`) in parallel to try out everything that is suggested here … and beyond.

For the moment it is proposed you rather follow the presentation of the trainer. But feel free to ask questions and suggest small detours, if you think it helps to understand what is discussed here.

Using Tcl Interactively

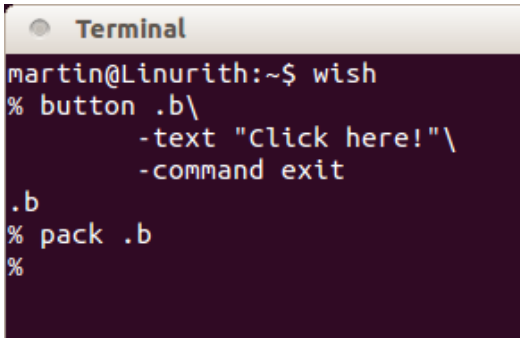
- The most basic way is this:
  - In a terminal window …
  - … execute command *tclsh*
- Voilà!



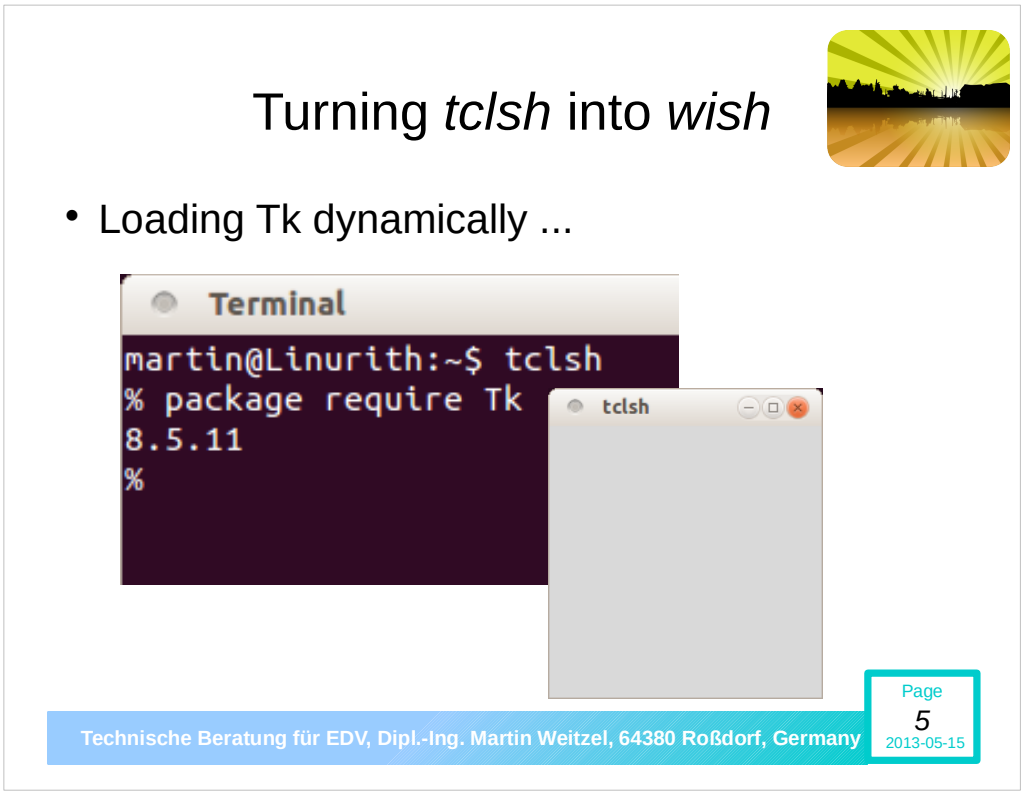
As the Tcl command language is so readily at hand, it is recommendable to try small examples while you read in as reference manual.

wish = Tcl with a GUI

- Try this
  - …
  - …
  - …
- … and:



The difference here is that a GUI window is opened, though it may sometimes be not so obvious, especially if you have a large screen and your window manager places it in corner far away from where your attention is ...



As the GUI window is actually an extension of the ordinary command language, in `tclsh` it can be even activated dynamically by loading a shared library.

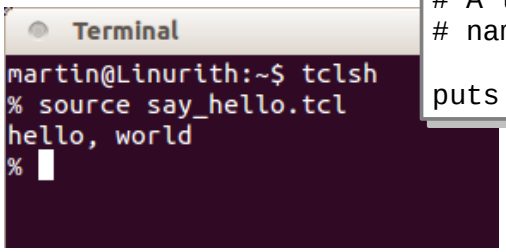
Note 1: The above behaviour became possible only after the `package` command was integrated in Tcl. In very old versions of Tcl and Tk there must be actually a different program be started.

Note 2: From a purely technical point of view it should be possible to start a Tk GUI window from any program that integrates Tcl as command language, like e.g. Vivado. In practice this option may be limited for the following reasons:

- When opening the GUI window Tcl may also try to run a file with commands to start-up and configure the Tk-part of Tcl completely. This may not be in the expected place or not be available at all. (Hint: under Linux the command `strace` might help to do the detective's work and find out what is missing and where it might be found, by comparing a tool like Vivado with a plain `tclsh` like above.
- Moreover, any program that integrates a Tcl command interpreter and besides this wants to use a Tk GUI window might need to merge two event loops. Depending if the use of Tk was already envisioned and properly prepared, this may be a trivial task, a moderately or very hard to solve problem, or practically impossible (at least without some sophisticated tweaking and recompiling).

Tcl Scripts

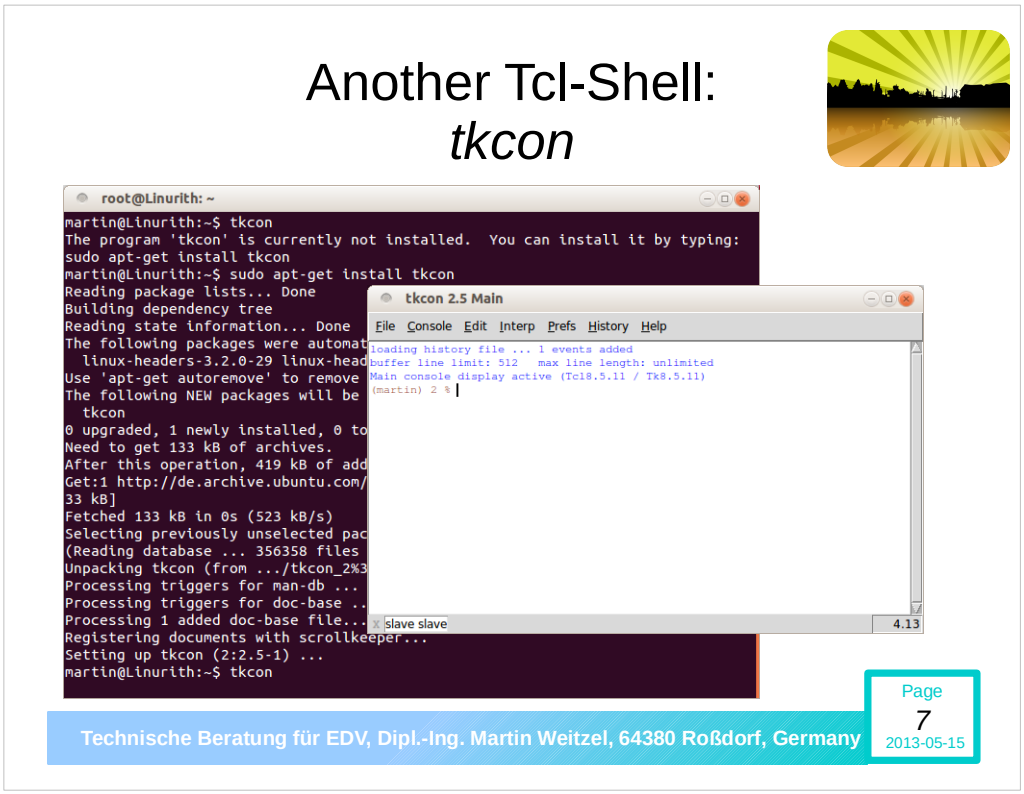
- Tcl-scripts are quite simple:
  - Write some valid Tcl commands into a text file
  - Let *tclsh* read the file and execute the commands
- Voilà!




```
# A text file with the
# name: say_hello.tcl

puts "hello, world"
```


So, maybe it's time now to open your laptops and go through the next few pages on your own.

Hello Tcl/Tk



The above also shows to install `tkcon` under Linux as an alternative to running `tclsh` and typing to it through a command window.

- `tkcon` may already have been installed on the laptops you use in this training.
- If you want to run Tcl/Tk on your own laptop under MS-Windows, `tkcon` will even be typically started by default when you execute `wish`. either from the command line or by clicking on the icon on your desktop or in the start menu.

A major advantage of using `tkcon` under Linux is much improved editing of the current and prior command lines – try it out by using the cursor keys!

Directly Executable Scripts

- Tcl-Scripts can be made directly executable
  - On Unix/Linux give execute permission and in the first script line add the name the interpreter
  - On MS-Windows connect the suffix `.tcl` with the interpreter

```
root@Linurith: ~
martin@Linurith:~$ chmod +x say_hello.tcl
martin@Linurith:~$ type tclsh
tclsh is hashed (/usr/bin/tclsh)
martin@Linurith:~$
```

```
#!/usr/bin/tclsh

puts "hello, world"
```


Another thing to try, if you like …

Tcl's Minimal Syntax

- This is all of Tcl's extremely simple syntax:
  - While obeying escape sequences and quoting ...
  - … find end of full command
  - … separate command into words
  - … substitute variable content and return values
  - finally execute built-in Tcl command or user-defined subroutine determined by first word ...
  - … handing over all following words as arguments.

… but now it's time to come down again from your first “free flight” and follow the presentation.

A Simple Count-Down

count_down.tcl

```
#!/usr/bin/tclsh
# ----------------------------------------
# count-down to zero
# ----------------------------------------

set x 10
while { $x > 0 } {
        puts -nonewline ".."; flush stdout
        after 1000 ;# delay for one second
        puts -nonewline [incr x -1]
}
puts ""
```


To save you some time to type the above text in an editor, you will find the code of this example (and of all the following too) in a directory the trainer will name to you.

Another Count-Down

count_down2.tcl

```
#!/usr/bin/tclsh

if { $argc > 0 } {
        set x [lindex $argv 0]
} else {
        set x 10
}
while { $x > 0 } {
        puts -nonewline ".."; flush stdout
        after 1000 ;# delay for one second
        puts -nonewline [incr x -1]
}
puts ""
```

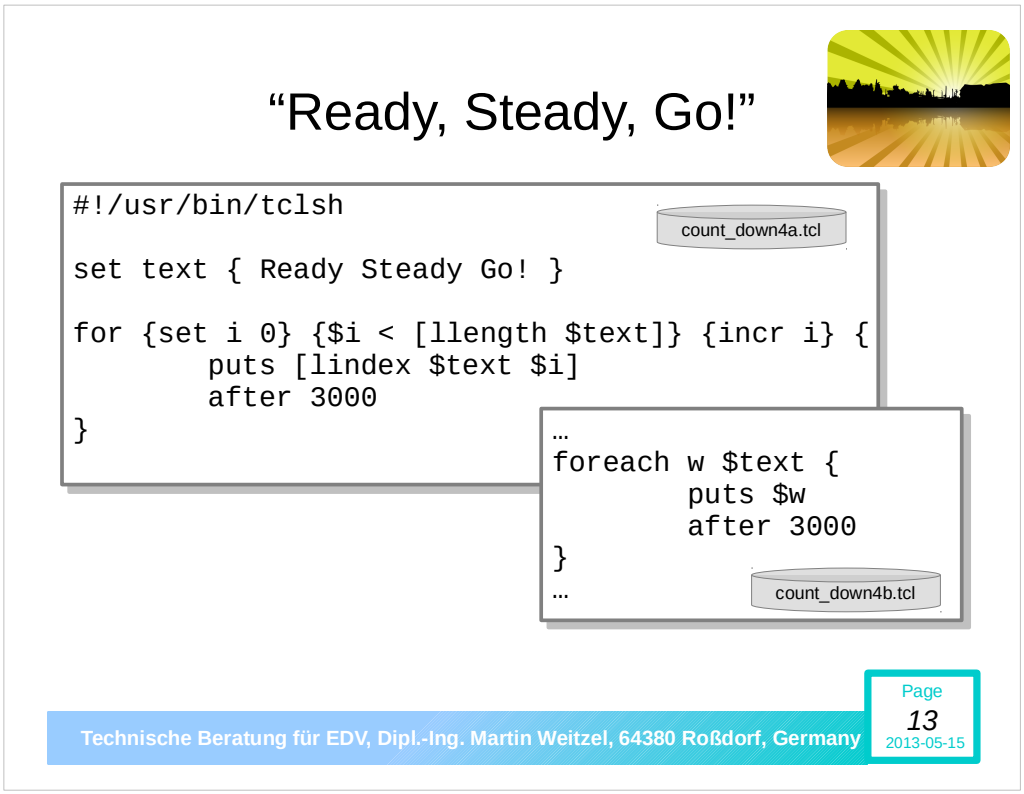

Spot the differences to the previous example!

And a third Count-Down

count_down3.tcl

```
#!/usr/bin/tclsh

if [catch {
        set x [expr 0+[lindex $argv 0]]
}] {
        set x 10
}
while { $x > 0 } {
        puts -nonewline ".."; flush stdout
        after 1000 ;# delay for one second
        puts -nonewline [incr x -1]
}
puts ""
```


Again: what has changed here compared to the previous example?



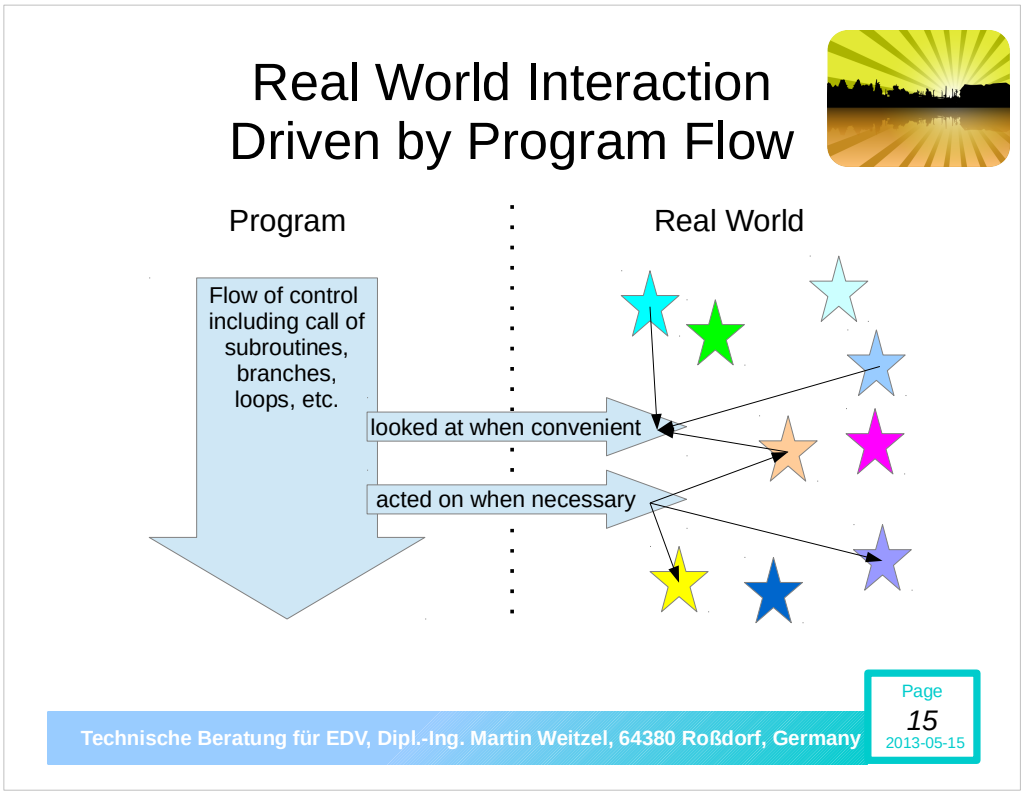
And now for this one – two variants this time, seemingly very different but with the same behaviour. (Or as it is said: there are many ways to skin the cat!)

A Simple GUI

count_down5.tcl

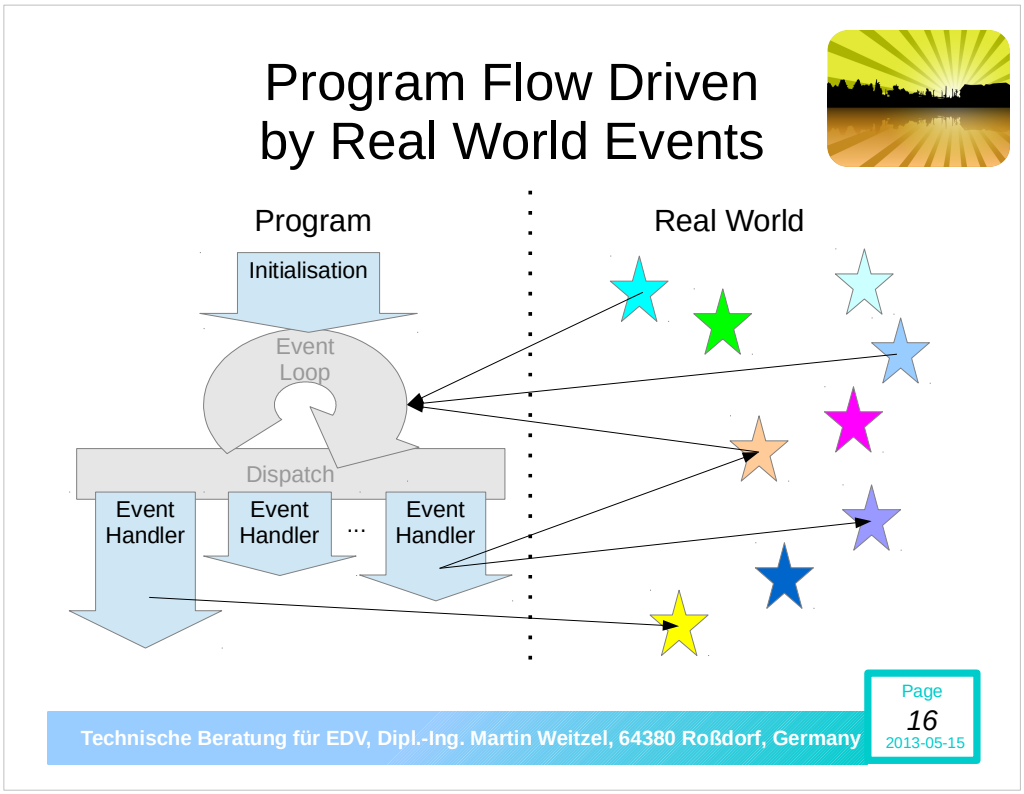
```
#!/usr/bin/wish

proc count_down {} {
        foreach w { Ready Steady Go! } {
                .b configure -text $w
                update
                after 1000
        }
        exit
}
button .b -text "Click to Start" -command count_down
pack .b -expand 1 -fill both
wm geometry . 250x250
```


As you see, GUI programming can be really neat and easy with Tcl/Tk!



But what actually is it, what makes the difference between a “command driven” program (`tclsh`) and one with a GUI (`wish`)?

Please make sure you really understand the above picture ...

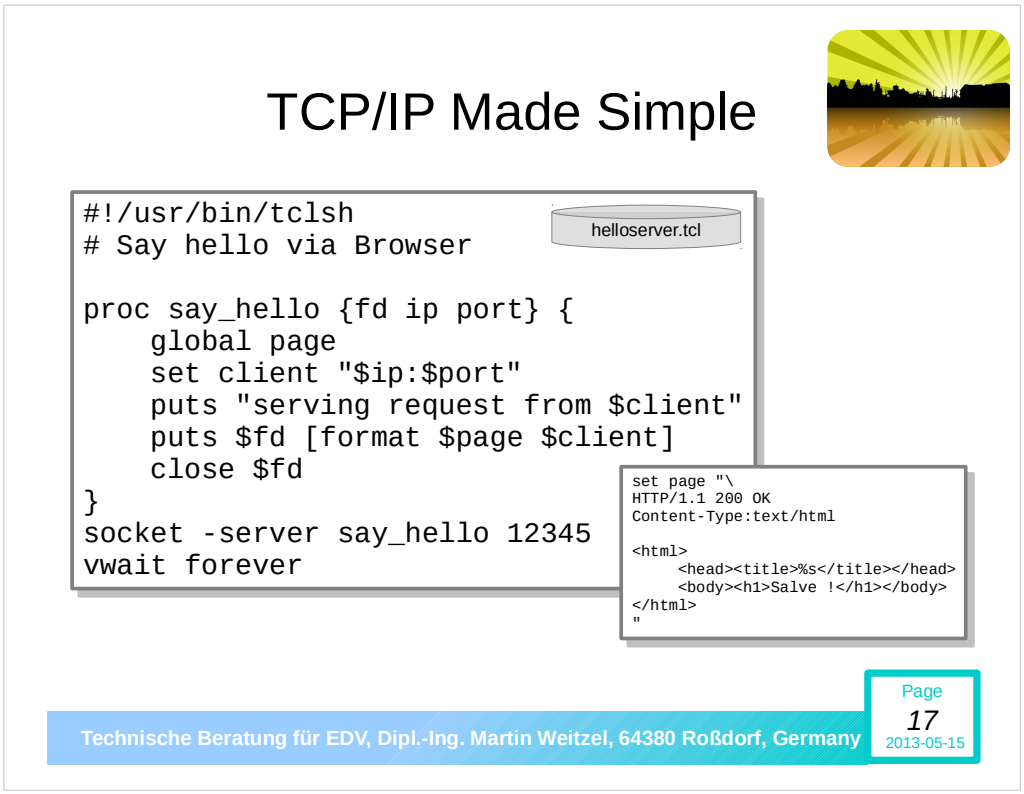


… and where it differs from this one.

Don't hesitate to fire more questions to the trainer until his explanations are sufficient to come to a proper understanding!

Note: There aren't any silly questions, silly answers at most but you will get no silly answer – but not in this training. it is really important you understand “who is in the driver's seat” as main difference between a design controlled by a prepared command script or by external events.

(Of minor importance but also of interest may be to consider the challenges that occur and must be solved when both approaches are mixed together; at least this were the base to understand how tools like Vivado might not only be driven by command scripts but in addition get a Tk GUI.)

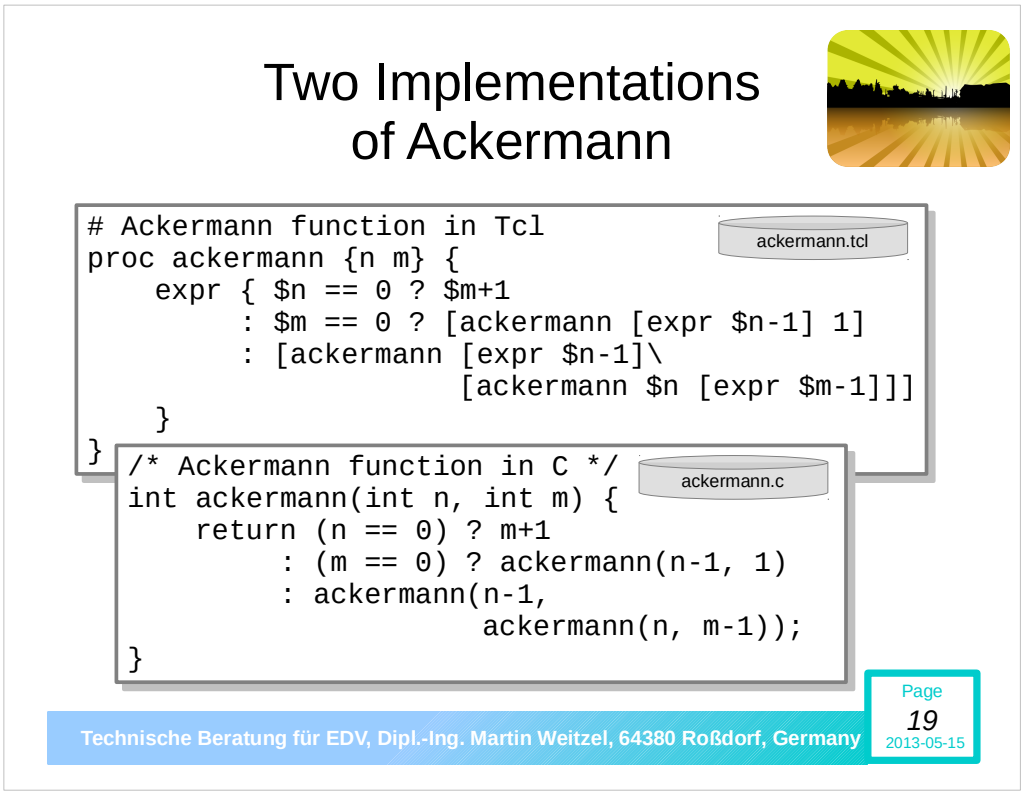


Would you have thought that a small TCP/IP-based “server” could be that simple?

Thinking “event driven” sometimes allows easy solutions for problems that seem hard to cope with at first glance – especially in a language like Tcl that embraces event driven designs at its core. Because it is that important in Tcl, a full later chapter is dedicated to the event driven approach.

Interfacing Tcl with C/C++

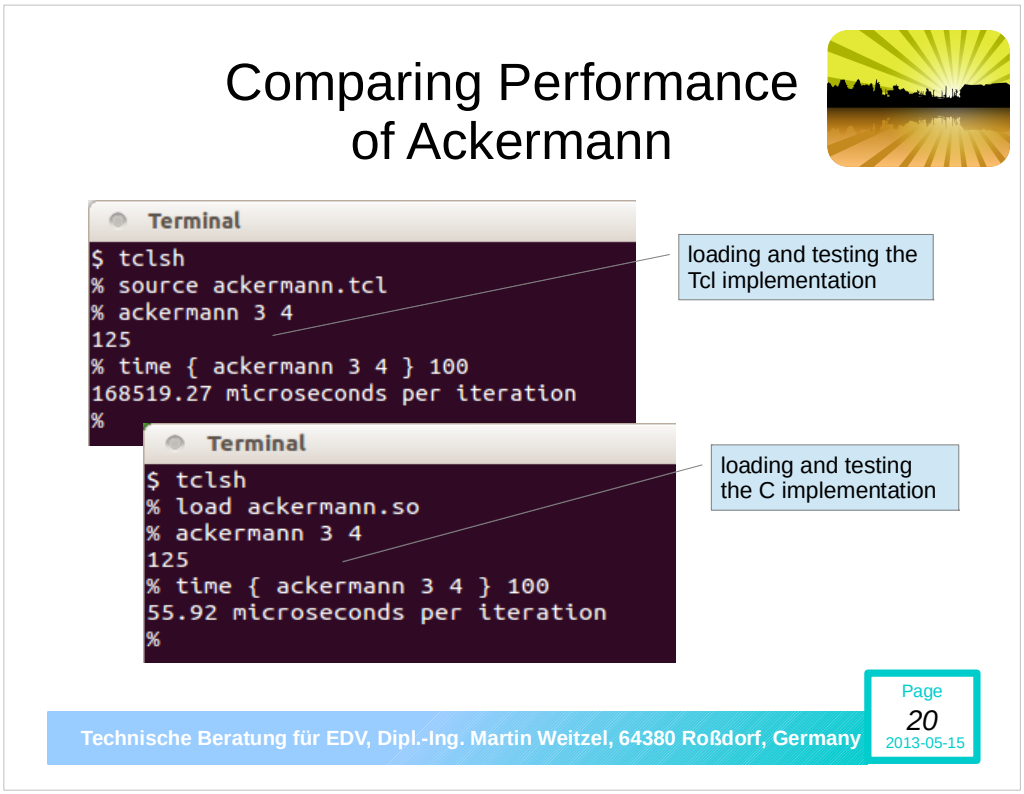
- Any subroutines may be implemented in C
  - Many built-in Tcl commands are implemented in C
  - Some may also be subroutines implemented in Tcl
  - Typical Tcl development adds more subroutines
  - There is the choice to implement these in Tcl or C
  - The former may be done by default …
  - … switching to the latter only if necessary to meet performance goals



If you still are typing to your laptops, please stop this now and follow the presentation.

The approach shown is a bit advance and you may never apply it in practice yourself, but it is at the heart of how Tcl commands may control arbitrary applications, like Vivado.

(After the next page time starts for practical exercises; you may very well go once more on yourself through the steps demonstrated by the trainer to show the effects of the above.)



The exact numbers may vary depending on the CPU power of your computer, but the trend should be always very similar to the above.

How to create the shared library is not further explained here, though it's no big deal and if you understand the C programming part a look into the `Makefile` supplied with the examples of this chapter may suffice to understand the details.

Time for some Practical Exercises

- As there has not yet been so much coverage of Tcl and Tk, you may now:
  - try-out once more the demos
  - ask questions if anything is still obscure
  - change some details if it helps to improve your understanding
- Of course, you may also lean back and just have some coffee or tee or juice and …

… *RELAX*

Note that during all following practical exercise you have the freedom to try Tcl and Tk on your own and there will always help be provided, if necessary

- Exercises should always be understood as suggestions!
- You may follow them exactly, closely, loosely, or not at all.

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Note that during all following practical exercise you have the freedom to try Tcl and Tk on your own and there will always help be provided, if necessary

- Exercises should always be understood as suggestions!
- You may follow them exactly, closely, loosely, or not at all.

Part 1
Tcl's Minimal Syntax

- What's ahead?
  - Commands separators
  - Word separators
  - Escape sequences
  - Variable substitution
  - Command Substitution
  - Quoting

The syntax of Tcl is simple.

The syntax of Tcl actually is simple.

The syntax of Tcl is actually very simple.

The syntax of Tcl is actually really very simple.

Or to speak with John Ousterhout – the creator of Tcl/Tk:

Some difficulties beginners may have stem from assuming a more complicated syntax as Tcl/Tk actually has.

21
et seqq.

Command Seperators

- There are two ways to separate commands:
  - Termination by end of line
  - Termination by semicolon

```
…
puts -nonewline $x
flush stdout
…
```

```
…
puts -nonewline $x; flush stdout
…
```


A later page covering *quoting* in Tcl shows how to hide a semicolon and avoid its interpretation as command separator.

Word Separators

- White space separates words in a command
  - Typically either a space character (ASCII 0x20) and a horizontal tab character (ASCII 0x9) is used
  - Any adjacent number of these characters is treated a single separator

```
puts -nonewline $x
```
three words

```
puts -nonewline          $x
```
three words too

```
      puts -nonewline   $x
```
also three words

A later page covering *quoting* in Tcl shows how to hide white space and avoid its interpretation as word separator.

Variable Substitution

- To insert the content of a variable use
  - … **$***variablename* … for plain variables, or
  - **$***arrayname***(***arrayindex***)** … for Tcl arrays
  - (more on variables/array in the next chapter)
- Note:
  - **Variable substitution does not nest!**
  - If a *var* contains the name of another variable, $$*var* will **not** induce a recursive substitution

A later page covering *quoting* in Tcl shows how to hide dollar signs and avoid they trigger variable substitution.

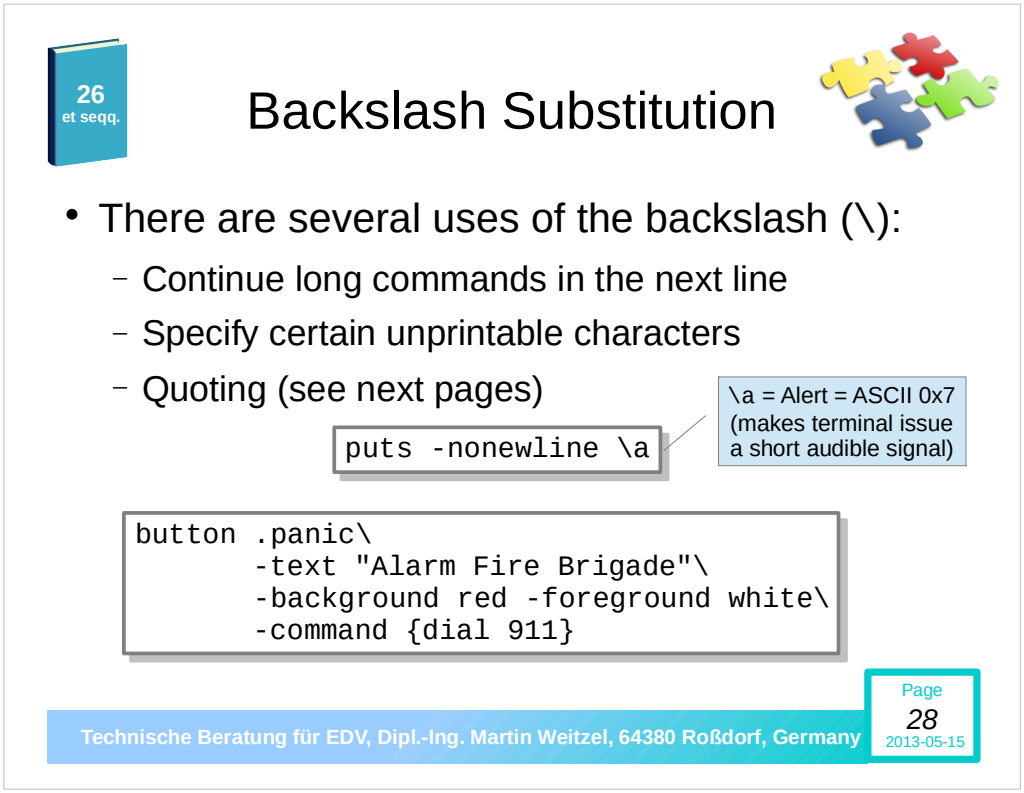
Command Substitution

- To insert the return value of a command use
  - … **[***any valid command***]** …
  - where a separate recursive syntax analysis takes place for the part in square brackets
- Note:
  - Command substitution nests!
  - … [*whatever* … **[*other command*]** …] … does what was (probably) intended

A later page covering *quoting* in Tcl shows how to hide dollar square brackets and avoid they trigger command substitution.

Note: if square brackets are not used nested, it is not necessary to hide a closing square bracket as by default it represents itself.

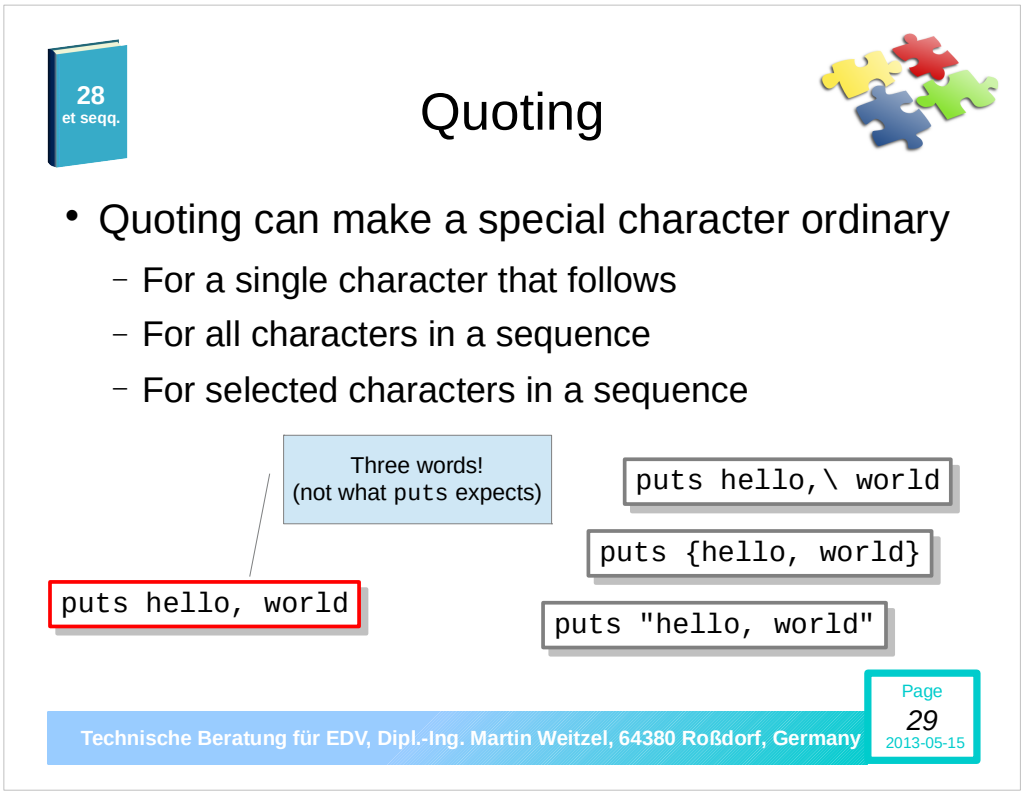
Q: Can you demonstrate the above with a trivial program?



Backslash substitution may also be used to avoid backslash substitution:

- Two backslashes in a row give a single backslash.
- Three backslashes in a row give a single backslash followed by backslash substitution.
- Four backslashes in a row give two literal backslashes.
- Five backslashes in a row give …

(… see you're dozing away, sorry!)

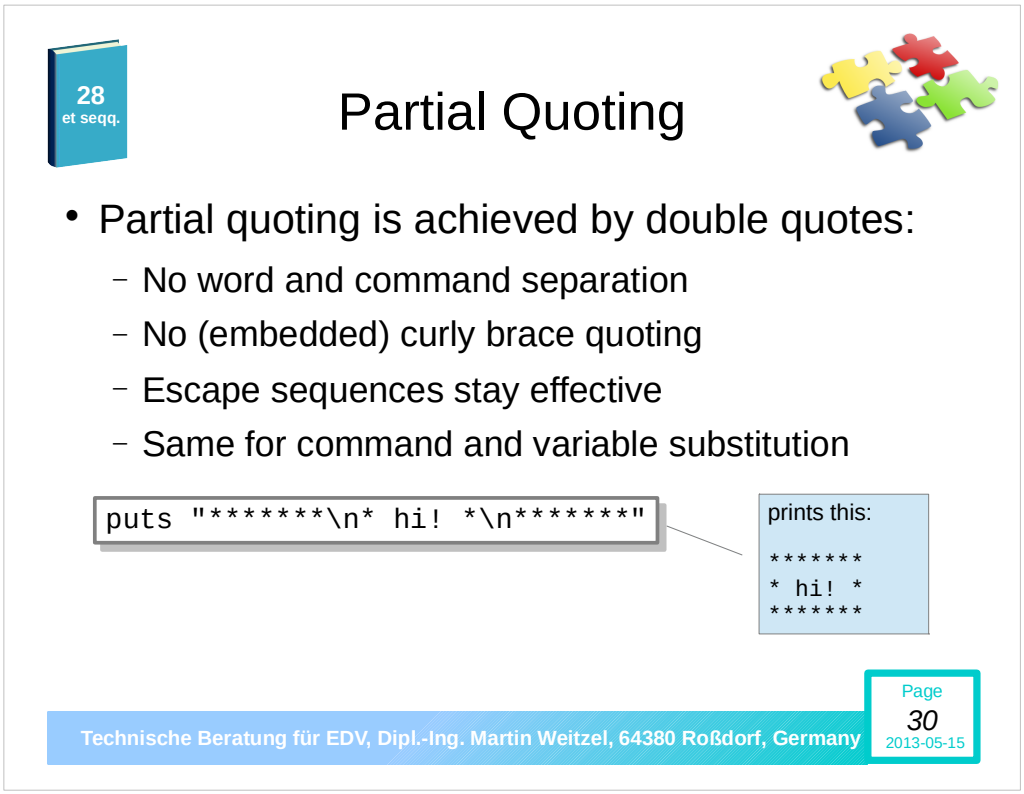


Backslashes cause special substitutions only if they are followed by certain characters.

In addition they can also be seen as quoting mechanism, as they remove the special meaning from any otherwise special character, like

- command separators
- word separators
- variable substitution
- command substitution
- themselves
- … (and some more what has not yet been covered)

meaning the character that follows remains literally (or “as is”, without the quoting backslash).



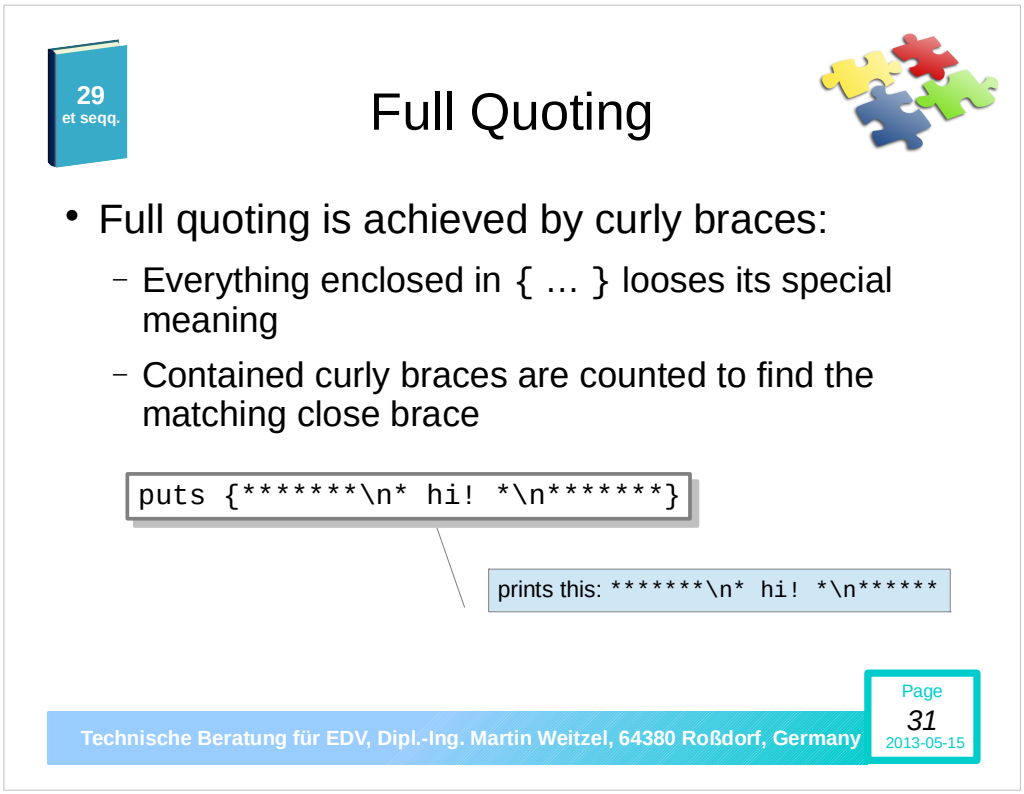
Besides quoting with backslashes, also double quotes may be used for quote (sic!) some text.

The rules might look slightly arcane at first glance but actually are very simple:

- Inside a pair of quotes any other character is quoted, except
  - semicolons and newlines as a command separators
  - white space as word separators
  - $-signs to trigger variable substitutions
  - square brackets to trigger command substitutions
  - backslashes to trigger other substitutions
  - backslashes to quote special characters
- and – because of the latter – also a backslash removes the special meaning from a double quote in double-quoted range.

As it is often the case with Tcl, you best run a tclsh (or `tkcon`) now and start some experimenting. Many-times the outcome will be what you expect – and if not there is usually a rational explanation, at least when you consider the alternatives, i.e. what were the consequences of the behaviour you first expected.

If you work in a group, this is a good opportunity to discuss the effects with your colleagues.



Finally, a quoted range may also extend from an opening curly brace to closing brace that matches the opening one.

Again, the details might look arcane at first but are really extremely simple:

- Nothing at all in a range delimited by curly braces is ever touched (in the meaning that it is changed) by the syntax interpretation.
- Contained opening or closing curly braces are simply counted
  - Counting up for an opening curly brace …
  - … and down for a closing one.
- When a closing curly brace is reached while the counter is – still or again – at zero, the quoted range terminates.
- Now the outermost curly braces are removed and the content is taken literally.

Be sure understand that a contained backslash does not really add any quoting – it is taken literally.

But if another curly brace follows adjacently, this one is <u>not</u> counted.

Q: How can you demonstrate the latter with some simple tests?

Comments

- A hash sign (#) turns a line into a comment
  - Only preceding white-space is allowed
  - Apart from this the hash sign must be the first character of the line

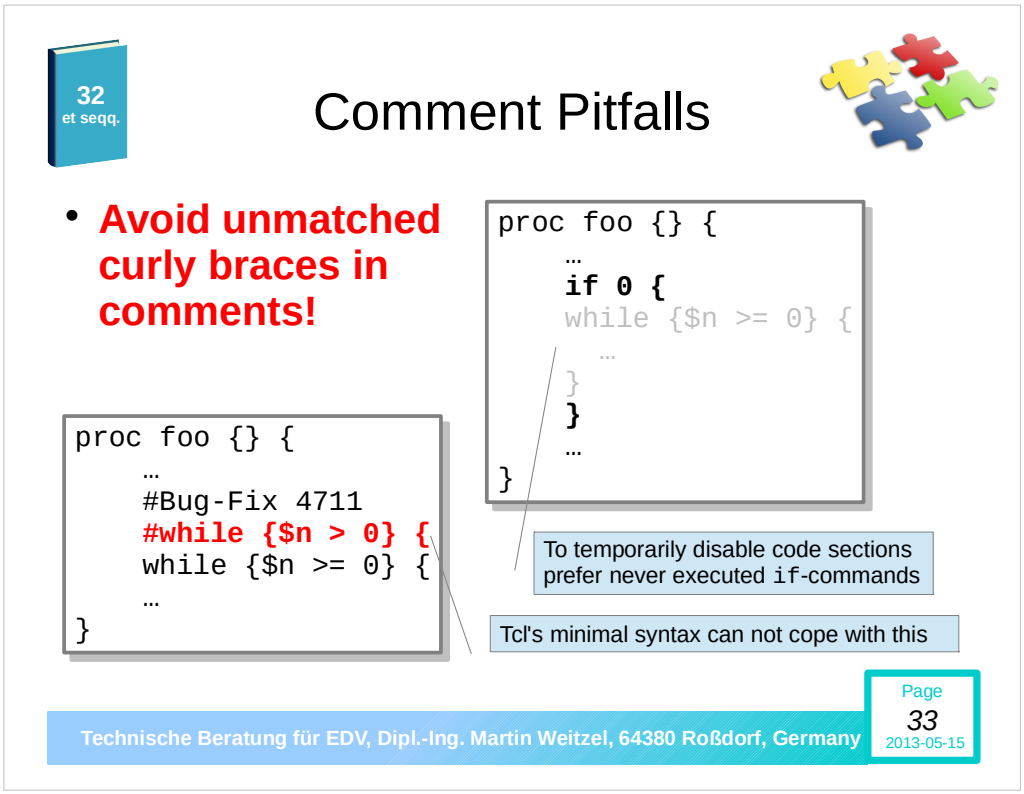
```
# this is a typical Tcl comment
    # this is an indented comment
set x 100 # this is NOT a comment
set x 100 ;# this IS a comment
# this is a comment that \
  extends to the next line
\# this is NOT a comment … but?
```


Comments are recognized only after quoting.

This has some consequences which may come to surprise, sometimes.

After having gone through the pitfalls demonstrated on the pages following, feel free to evaluate the design choices that have obviously been made in Tcl and consider the consequences resulting from a different design, one that tries to avoid the pitfalls …

May be you will be surprised how much complexity any “improvement” in this area will actually add to the syntax interpretation.



Avoid unmatched curly braces in comments.

Avoid unmatched curly braces in comments.

Avoid unmatched curly braces in comments.

Avoid unmatched curly braces in comments.

Avoid unmatched curly braces in comments.

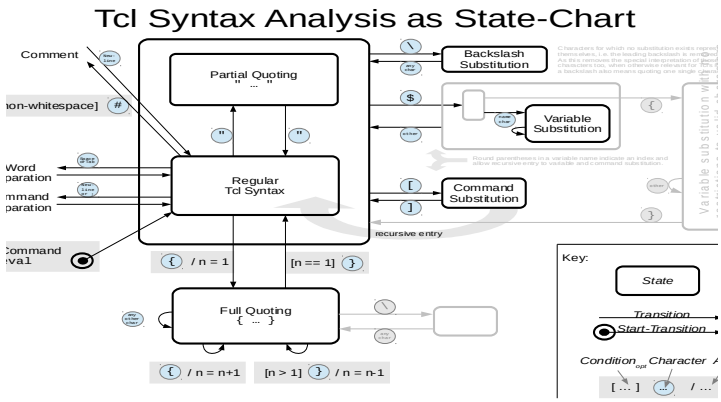
Avoid unmatched curly braces in comments.

Avoid unmatched curly braces in comments.

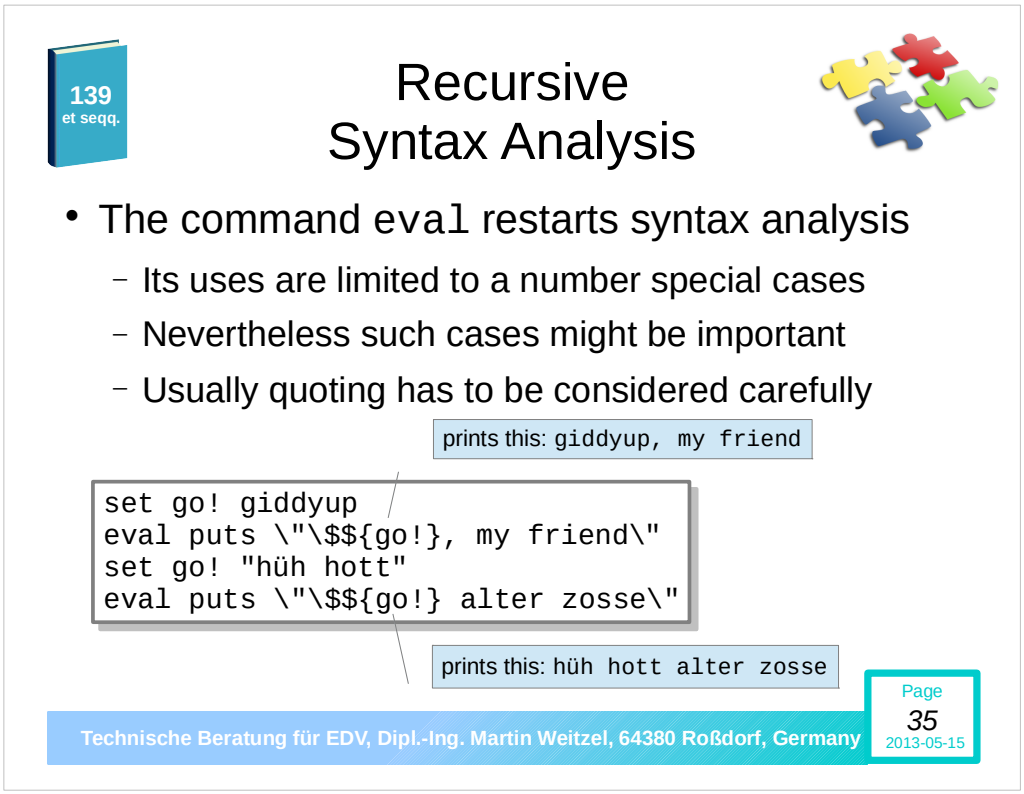
(Well, now, either you've got it or you will get lost sometimes when programming with Tcl.)

Putting it All Together

- You should also have received a personalized DIN A4 version



It's a good time now to remind the trainer to hand you our personalized copy of the above.



To be honest, at this point the author of this lines once came to the point where he thought:

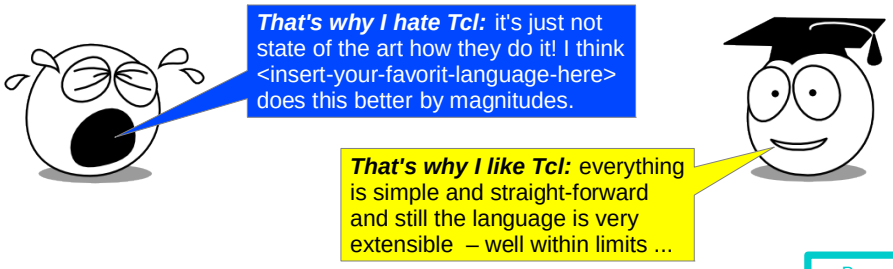
Oh, how is all of this difficult in Tcl, I think I'll never get it, it blows my mind.

This lasted some hours or days (don't actually remember, it was nearly twenty years in the past), but a few days later he thought

How could I ever think I'll never get it … it's sooo simple and straight-forward!

Syntax Revisited: The Full Story

- This is it what makes Tcl somewhat different:
  - Its syntax is rather minimal because
  - many “interesting things” are in its commands, e.g.
  - arithmetic expressions evaluation and flow control



Many Tcl users sooner or later take their position in this respect.

You will reduce your hassles with Tcl substantially if you try to get on the right side.

(You need not come to love Tcl for its syntax, but stop fighting it, even if it's not your first choice.)

Time for some Practical Exercises

- Actually, we have not covered that much land so far that you can write a neat program, instead
  - try to develop and improve your understanding for Tcl's syntax with trivial examples
  - especially try how those mechanisms nest ...
  - .. how quoting nests etc.
- And in between your experiments lean back sometimes and just …

… *RELAX*

In case you want to try variable substitution, you may first want to set some variables:

- `set x 42`
- `set z "hello"`
- `set z "hello, world"`

Q: Why is quoting necessary in the last case but could actually be omitted in the second last?

To try command substitution you also may use the set command. e.g.:

- `puts [set z]`

Otherwise your command inputs (for `tclsh` or `tkcon`) may look as simple as:

- `puts "{"`
- `puts {"}`
- `puts "{"`
- `puts \{`
- `puts {{}}`
- `puts {\{}`
- `puts $x`
- `puts $y`
- `puts $z`
- `puts {$z}`
- `puts {[set z]}`

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

If you have any suggestions how the

“Tcl Syntax as State Chart”

diagram could be improved – without blowing the space of a DIN A4 page – let us know.

Part 2
Data Structures

- What's ahead?
  - Plain variables
  - Arrays
  - Lists
  - Dictionaries
  - Namespaces

Different data structures offer different ways to bundle simple (plain) variables together. This makes sense when the bundled variables serve a similar purpose. Some data structures may also contain nested data structures as parts.

To give an example with a concrete background, consider an entry in a log-file. It may consist of

- a severity
- a numerical error id
- some descriptive text
- a time-stamp
- …

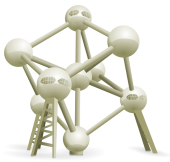
The time-stamp may in turn have a sub-structure, say

- day, month, year
- hour, minute, second

In non-object-oriented designs data structures are in principle independent from their processing, which is often encapsulated into routines (or functions). But often data structures and their processing helpers reflect a similar hierarchy. In the above example there may be helper functions to extract fields from log-entries; and to compare time-stamps there may be other helpers that know the internal details of the date sub-structure, while as part of the log-entry a times stamp may be considered as a single (technically opaque) unit.

41
et seqq.

Plain Variables



- Variables in Tcl are typically
  - *defined* by assigning a value with `set`
  - *accessed* by prefixing the name with a $-sign

```
set temperature 26
```

The $ is not part of the variables name, so be careful not to call a variable "*dollar*-something". It's quite simple: You use the $ to indicate you want to access the value, but you never use it when you need to change the value!

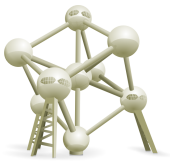
```
puts "water is $temperature centigrade today"
```


The effect of the `set` command depends on the number of its arguments:

- The first argument is always taken as name of the variable.
- The second is optional and if present gives the value to store in the variable.
- In any case the return value of the command is the value of the variable
  - either the new one, if there is a second argument,
  - or the current one if there is only one argument.

41
et seqq.

Unsetting Variables



- The `unset` command removes variables
- Accessing an unset variable is an error
  - It is possible to test whether a variable exists
  - Alternatively access errors may be caught

```
if [info exists x] {
    unset x
}
…
```

```
catch { unset x }
…
```

```
# since Tcl 8.4
unset -f x
```

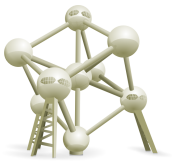
```
if [info exists x] {
    puts "x=$x"
}
```

Pythonists may call this LBYL style (look before you leap) versus EAFP style (easier to forgivingness than permission)

Note that like it is an error to access the value of a variable that has not yet been set – be it with a `$`-substitution or using the `set` command with a single argument – the attempt to `unset` a variable not previously set is an error in Tcl and a special technique is required to avoid that error if a variable is to be unset that is not surely set.

43
et seqq.

Arrays (1)



- Tcl arrays
  - have an index enclosed in parentheses
  - no restrictions with respect to their index
  - which – of course – includes numeric indexing

```
set temperature(ambient) 27.4
set temperature(pool) 25.3
```

You also program in C/C++ sometimes? Do not compare Tcl arrays to the built-in arrays of these languages. Tcl arrays are more like STL-maps … or what is sometimes called “hashes“ in other interpreted languages.

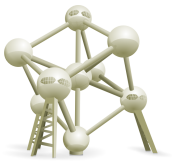
Like the name of a variable can be literally anything – as long as quoting is applied accordingly when the value is set or accessed – also the index can be anything.

Of course, for using arrays as classical “containers with enumerated cells”, the index should be numerical, though it need not be strictly sequential. (In other words: Tcl arrays are already sparse arrays by their nature.)

Besides the “array-like” use Tcl arrays are often used to emulate *Structures* (C nomenclature) or *Records* (e.g. Pascal), which were never directly supported in Tcl until dictionaries were introduced (covered later).

43
et seqq.

Arrays (2)

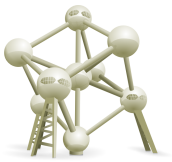


- The command `array` is a handy helper:
  - Besides initializing an entire array it allows
  - to find out the number of elements
  - write loops to access each element
  - and much more

```
puts "[array size temperature] measurements taken:"
foreach location [array names temperature] {
        puts "$location: temperature($location)"
}
```


The `array` command is actually a “wrapper” around a set of sub-commands that access arrays as a whole or supply certain common operations with arrays, like iterating over all its elements.

Lists (1)



- Tcl lists are
  - sequential containers for values and
  - often initialized with quoted word sequences ...
  - … or the `list` command in the more general case

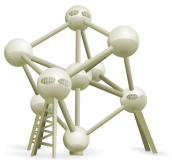
```
set words {Ready Steady Go!}
set cities [list Berlin "New York" Paris]
```

You also program in C/C++ sometimes? It's the Tcl list – not the Tcl array(!) – that comes close to built-in arrays of C/C++ (… or rather the STL vector, as Tcl lists automatically grow if more elements are added.)

A Tcl `list` is very similar to a C-style array. If indices start with zero and increase consecutively, access to elements in a Tcl list is much more efficient then access to elements in a Tcl array.

101
et seqq.

Lists (2)



- Tcl lists are
  - manipulated with special commands,
  - which typically begin with the letter `l` (ell)

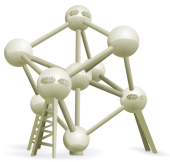
```
puts [llength $words]
puts [lindex $words 0]
```

prints this:
3
Ready

```
puts [llength $cities]
puts [lindex $cities 1]
```

prints this:
3
New York

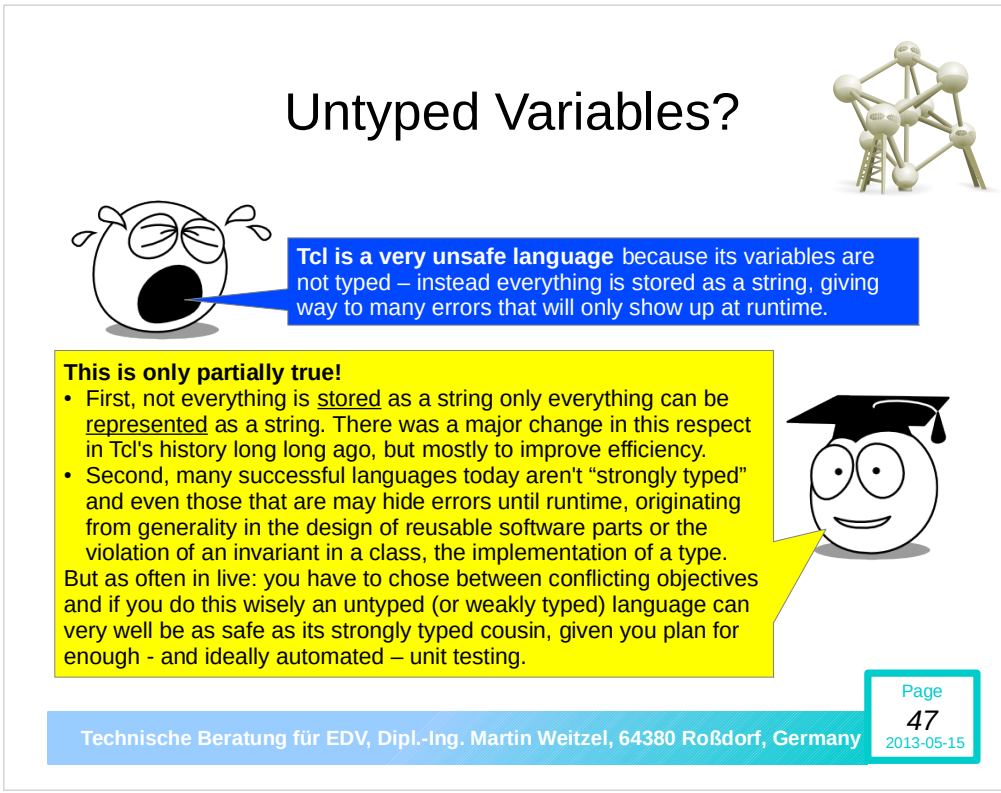
Dictionaries



- Tcl dictionaries are two-faced:
  - At first glance they look like key-value pairs ...
  - … but actually are a recursive tree structure ...
  - … of unlimited depth
  - All access and manipulation is through the `dict` command (or rather its sub-commands)

Ever heard of *Design Patterns*?
As introduced in the “GoF-book”?
You know what a *composite* is good for?
Use a dictionary if you need a composite in Tcl!

Dictionaries are a relatively recent addition to Tcl and may hence not be found in all the places where they might be used. Especially in old code with a long tradition, the use of combinations from Tcl `list`-s and `array`-s will be found where Tcl `dict`-s had clear advantages.

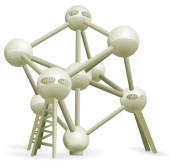


Any variable may store anything and there are text representations for any Tcl data structure that can be used to convert back and forth, if necessary. (Nearly so, the exception are Tcl arrays, but here this proposition applies to all of its elements.)

Despite the fact that in Tcl still "anything can be represented as string of characters", since long the internal representation is chosen for best efficiency, depending on the actual content. This makes different types of variables (and data structures like `list`-s and `dict`-s) efficient in Tcl, as it avoids the many "back-and-forth"-conversions old Tcl versions had to perform.

53 et seqq.

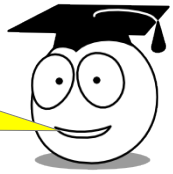
Arithmetic Evaluation



- Any variable that contains a numeric value
  - may participate in arithmetic evaluation
  - induced by the special command `expr`

```
set x [expr $x+1]
```

```
set c [expr sqrt($a*$a + $b*$b)]
```



Byte code in modern Tcl can might be created in a slightly better performaning variant if you add a level of quoting:
- … `[expr {$x + 1}]` …
- … `[expr {sqrt($a*$a + $b*$b)}]` …

The internal representation for a variable to hold arithmetic values depends on its initialiser:

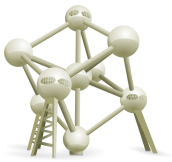
- if it looks like an integral value, it stored as integer (usually in the most efficient integral type the hardware can offer);
- if it looks like a floating point value, it is stored as such (often as a double precision IEEE 754 type with 64 bit, i.e. 52 bit mantissa and 11 bit exponent).

Furthermore for recent Tcl version the following is true:

- if the initialiser looks like a “long” integral number – i.e. an integral number with more digits as a the hardware integer type can hold of suffixed with the letter `L` – an “unlimited precision” integral type is used.

As detailed on the next page, when combining variables that hold arithmetic values in expressions, the “largest type” will be determined that can hold the result without (or only little) loss.

Integral or Floating Point

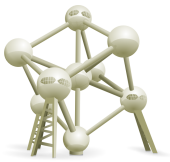


- Details of `expr` depends on the operands:
  - **first** a “widening” of the operands takes place, much like in C/C++, with a ranking from
  - `int` – a “native” built-in integral type, at least 32 bit signed, to
  - `long` – an unlimited precision integral type, to
  - `float`-ing point (machine dependent),
  - **then** the result type is chosen according to the largest operand type within that ranking

Q: What is the result type if some floating point value with a fraction and a sufficiently large “long” are combined in an arithmetic expression, so that neither a floating point nor a long could hold the result without any loss?

47
et seqq.

More Arithmetic Contexts (1)



- Beside `expr` other arithmetic contexts exists:
  - Incrementing/decrementing an integral value ...
  - Accessing a list from its end …
  - … (and surely more) ...

```
set x [expr $x+1]
```

```
incr x 1
```

```
incr x
```

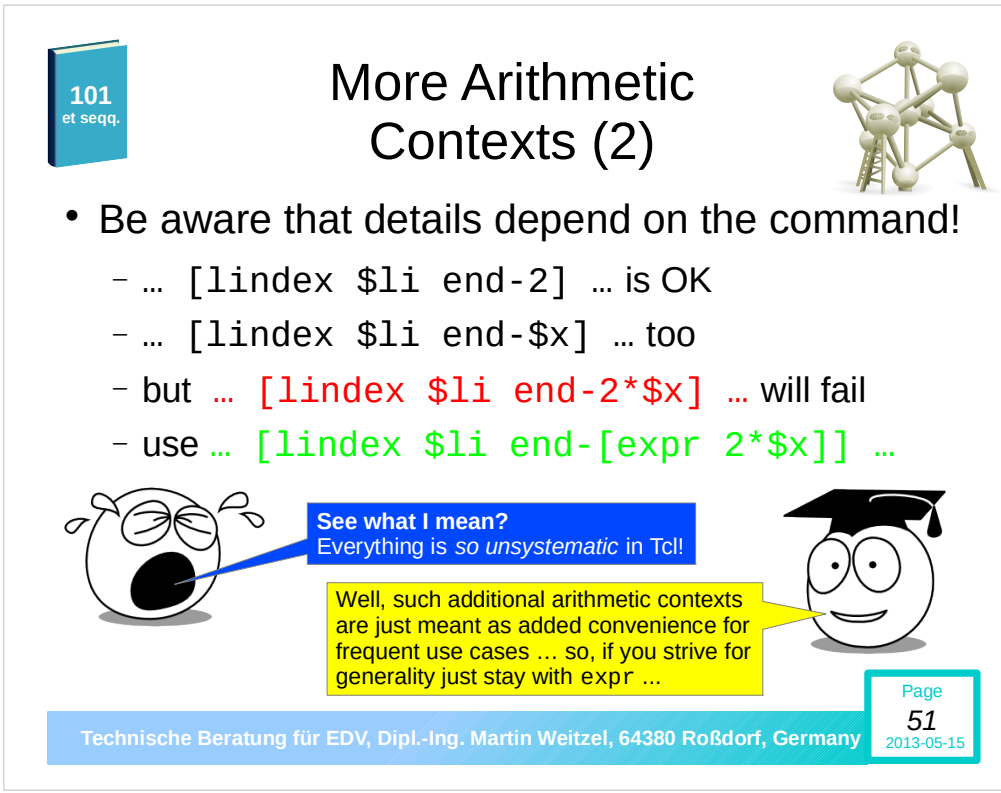
```
set y [expr $y-$x]
```

```
incr y -$x
```

```
… [lindex $li [expr [llength $li]-2]] …
```

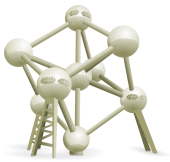
```
… [lindex $li end-1] …
```


Q: What happens it the largest value that can be stored in a 32 or 64 bit integer – depending on what your Tcl version uses to store non-long integral values – gets incremented by one?



There are lesser problems with this as usually expected, as such extensions take most typical use patterns in account and errors typically show up early. Pitfalls mostly occur when back-porting code using such extensions to be run with an older version of Tcl, not yet supporting the extension.

Namespaces



- Using namespaces has the following benefits:
  - Reduced danger of accidental name clashes
  - Unambiguous references from everywhere but short names for internal use

```
eval namespace GUI {
    proc init {} { … }
    proc startUp {} {
        init
        …
    }
}
```

```
eval namespace Other {
    proc init {} { … }
}
```

```
eval namespace Mine {
    proc foo {} { GUI::init }
    …
}
```


While small project will usually not use this feature, when working with Tcl as command language embedded in a given framework, the use of certain namespace may be required. (Of course, this will be documented then and also visible from examples.)

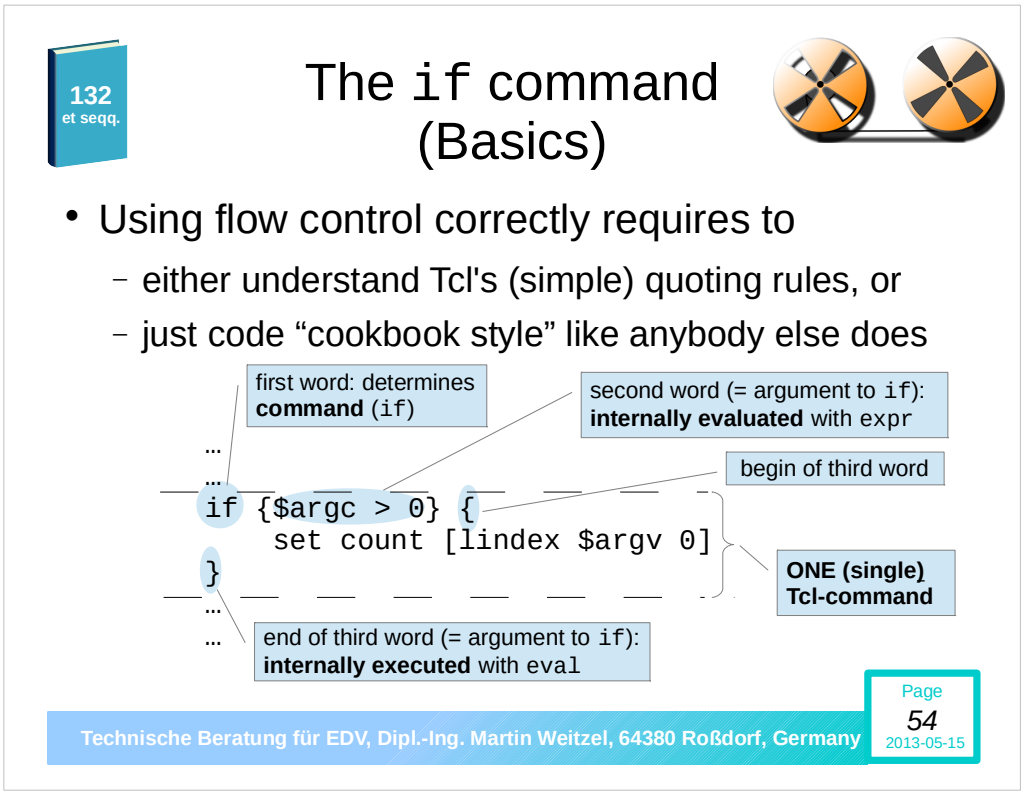
Part 3
Flow Control

- What's ahead?
  - Branches
  - Loops
  - Exceptions

The most important thing you should understand here probably is:

- All of the Tcl flow control is implemented by ordinary “commands” which simply apply special evaluation on their arguments, like
  - evaluating an expression for its “truth” value:
    - zero is “false”
    - anything else is “true”
  - or (re-) evaluating it as Tcl command – so it should have been put into curly brace quotes in the first place.

(Maybe the above comes at the right time to prepare your mind or too early – i.e. you do not understand what it says right now; don't worry: you will understand it after we ran through that chapter and you review this page once more.)



How exactly you chose to quote the argument in the example above does not matter at all – Tcl's `if` command will **not** see it at all! (Think about it!!)

So, the following is equivalent to the above:

```
• if \$argc\ >\ 0 {
      set count [lindex $argv 0]
  }
```

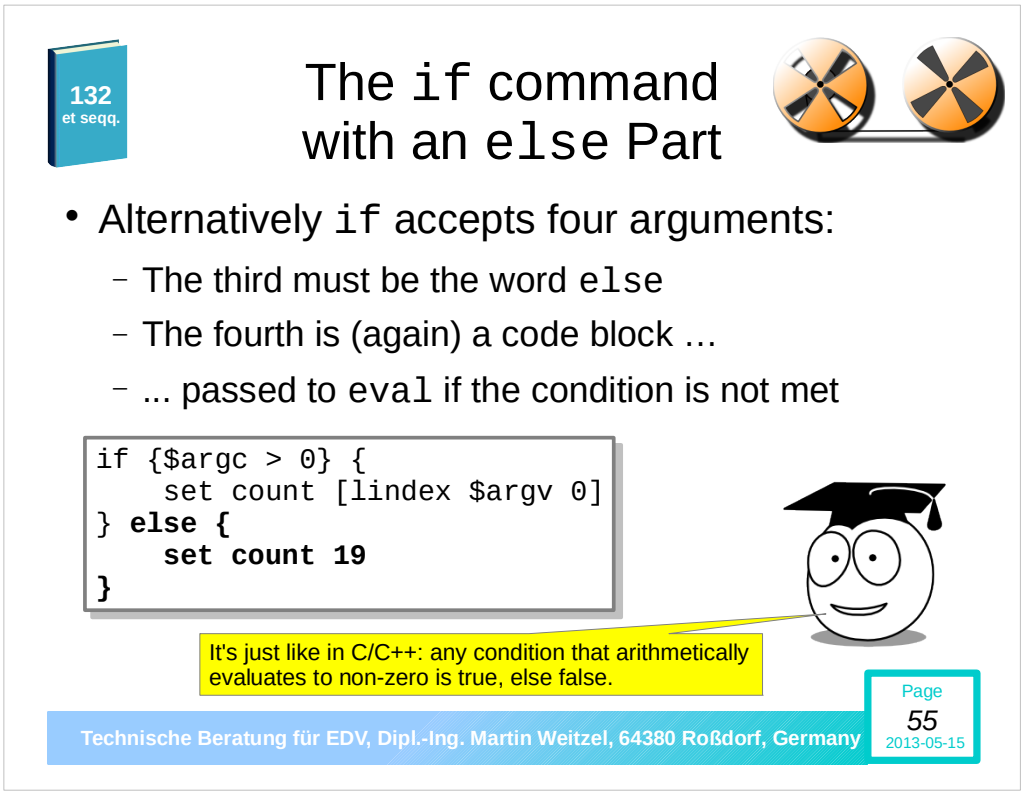
Or:

```
• if {$argc > 0} "set count \[index \$argv 0]”
```

But not (quite) to:

```
• if \$argc>0 "set count [index $argv 0]"
```

Q: Why (why not)?



You can only throw in a `then`:

```
if {$argc > 0} then {
    set count [lindex $argv 0]
} {
    set count 19
}
```

While lazy natures might like to mitt the `else`::

```
if {$argc > 0} {
    set count [lindex $argv 0]
} {
    set count 19
}
```

Also this one-liner works (but is probably less clear):

```
if {$argc > 0} {set count [lindex $argv 0]} {set count 19}
```

Note also that Tcl's `expr` command can evaluate conditional expressions (like C does), hence if you need only to select between too values to assign to a variable, you can avoid the `if` command altogether:

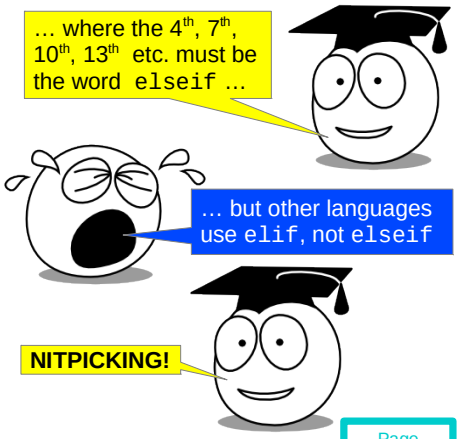
```
set count [expr ($argc > 0) ? [lindex $argv 0] : 19]
```

132
et seqq.

Chaining of `if - else`

- Actually `if` accepts even more arguments ...

```
if {condition-1} {
    first-codeblock
} elseif {condition-2} {
    second-codeblock
} elseif {condition-3} {
    third-codeblock
} elseif {condition-4} {
    fourth-codeblock
else {
    last-codeblock
}
```



Do you like nit-picking? Feel free to join in.

(Or get off from Tcl before we actually take up speed – last chance now!)

133
et seqq.

Multiway Branches

- The `switch` command allows multiway-branches with case (label) comparisons
  - being exact,
  - “glob-style”, or
  - via regular expression


```
switch -regex $input\
{-?[0-9]+\.[0-9]*([Ee][+-]?[0-9]+)?} {
    …
} {-?[0-9]+} {
    …
}
```

float number input

integral number input

Probably the `switch` command is the most cumbersome to fit as flow control structure with Tcl's scarce syntax.

Watch in the examples following – in the presentation and in example code – for an alternate form of this command which may sometimes be clearer (but has its drawbacks too).

136
et seqq.

Commands for Repetitions

- The following commands are for loops:
  - `while` (much like if but with repetition)
  - `for` (may be rewritten as while)
  - `foreach` (process each element of a list)

```
for {set i 0} {$i < 100} {incr i} {
    …
}
```

```
set i 0
while {$i < 100} {
    …
    incr i
}
```


A very important thing to understand is that the following two have the same effect

- `if {$a > 0} …`
- `if "$a > 0" …`

but this is **not** true here:

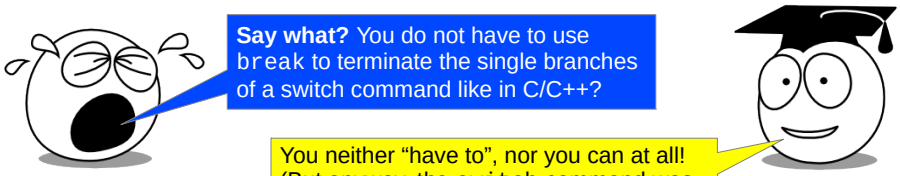
- `while {$a > 0} …`
- `while "$a > 0" …`

Q: Explain!

136
et seqq.

Premature Restart/ Break-Out from Loops

- The following can be used only inside loops:
  - `break` (branch to end of loop)
  - `continue` (re-evaluate condition for `while` or increment and condition for `for` commands)
  - `return` (covered in the next chapter)



So, you cannot have fall-through behavior, even if you would like to have it!

Escaping from Errors

- The command error allows to
  - escape from an erroneous context
  - to an unknown destination

```
switch -glob $filename {
    *.c   { set cfile $filename }
    *.cpp |
    *.cxx { set cppfile $filename }
    *.tcl { set tclfile $filename }
    *     { error "unsupported file type" }
}
```


In comparison with a programming language that supports control flow branching via exceptions, like C++, Java, etc., Tcl's `error` command is equivalent to a throw-statement.

Comparing Tcl to C, the closest similarity is to `longjmp` though the internal working of the latter is much more basic and close to the hardware.

Q: When you compare the above to the example introducing Tcl's `switch` command, can you spot a – very subtle but important – difference?

Regaining Control after Errors

- The command `catch` may eventually regain control after an error occurred

```
set errs 0
foreach name $argv {
    if {[catch { openfile $name } msg]} {
        puts stderr "$name: $msg"; incr errs
    }
}
if {$errs} {
    puts stderr "$errs error(s) cause exit"
    exit 1
}
…
```

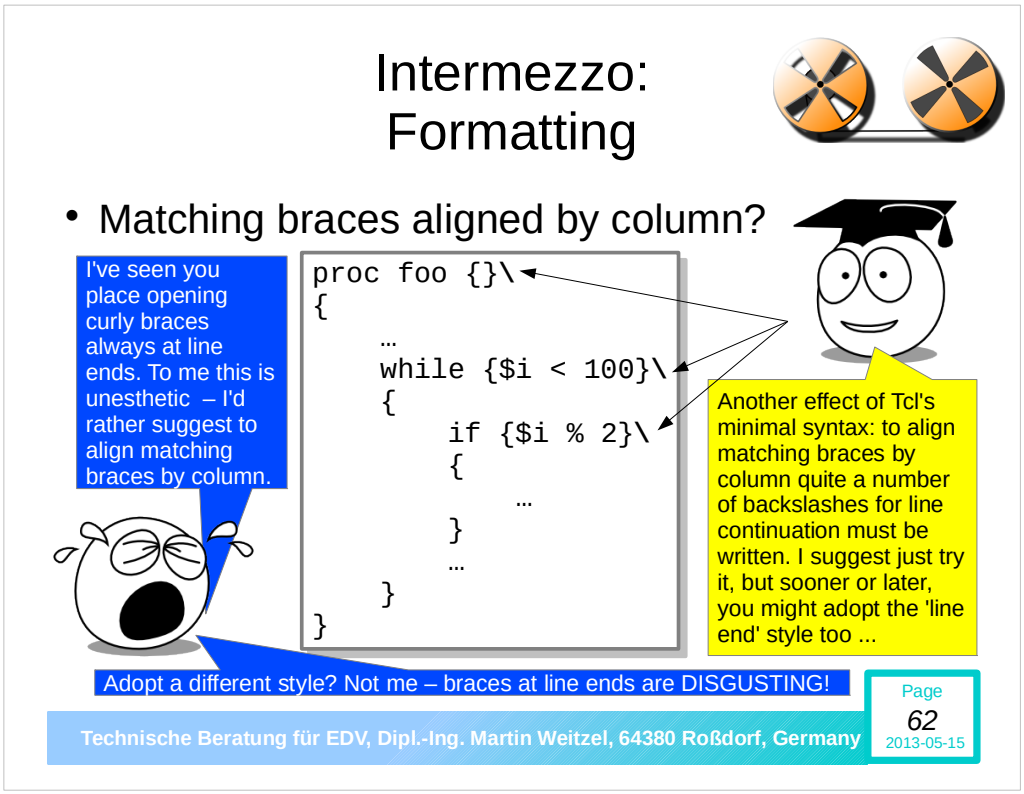

In comparison with a programming language that supports control flow branching via exceptions, like C++, Java, etc., Tcl's `catch` command is equivalent to a catch-block in a try-statement.

Comparing Tcl to C, the closest similarity is to `setjmp` though the internal working of the latter is much more basic and close to the hardware.

Note that the return value of a catch command often needs to be tested for branching between the error case and a successful completion.

Hence Tcl's `catch` command is typically executed

- in a command substitution which
- in turn is evaluated as `if`-condition.



Watch the quarrel end enjoy. (And to avoid more tears: don't fight with Tcl – take it as it is.)

Time for some Practical Exercises

- Write a program that prints the first ten
  - squares
  - square roots
  - prime numbers
  - numbers of the Fibonacci series
  - …
- Change the program so that
  - the upper limit can be set with a command line argument,
  - using the value 10 as the default if none is given.

If you have no good idea how to start, take a look at page 10 (first chapter) but reverse the loop direction.

For the actual calculation,

- determining the square should pose no problem (hopefully);
- for determining the square root look at the documentation for Tcl's expr command, especially the supported mathematical functions;
- calculating primes in addition
  - needs a data structure (an array or a list) to store the primes calculated so far,
  - another (embedded) loop to check prime candidates against known primes,
  - and – maybe – you need the following hint:
    you can test whether two numbers divide evenly by checking the result of the modulo operation (%) for zero;
- if you don't know the Fibonacci definition:
  - its first two numbers are `1` and `1`,
  - then successors are the sum of their two immediate predecessors.

If you need an idea how to tackle the second major step, i.e. controlling the program with a command line argument, have a look at page 57 and adapt it accordingly

(If you are very quick with both solutions, you may want to try several of the alternatives suggested on the lower half of page 57.)

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Honestly now: did you already adopt the “asymmetric bracing style” or still use line continuations after branch and loop conditions?

Part 4
Subroutines

- What's ahead?
  - Defining subroutines
  - Calling subroutines
  - Handing over arguments
  - Returning values

In a nutshell:

While some programming languages make a difference between plain S*ubroutines* and *Functions* returning a value, this is not the case in Tcl.

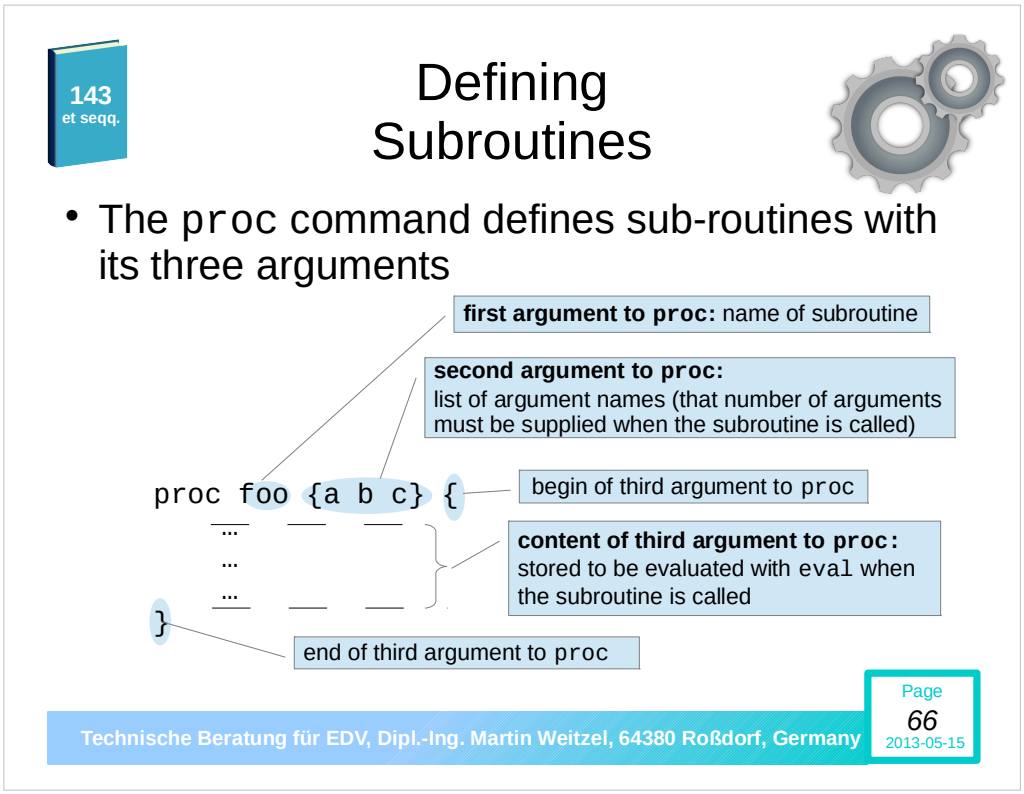
Each subroutine returns a value, either explicitly or implicitly:

- a value is explicitly return with the `return` command;
- if there occurs no explicit return before the end of the function is reached, the function returns whatever the last command or function it had executed has returned.

Note that functions may not only return with a `return` command or after their last command has been executed, also an `error` command terminates execution of the currently active function.

Furthermore, there may be function arguments, but according to the untyped nature of Tcl variables

- on a function call only the number of arguments are checked, not their type,
- but of course a function has the freedom to check the arguments it received and return an error if there are some constraints which are not met.



Make sure you have understood that Tcl's `proc` command itself is kind of a function which requires exactly three arguments.

145
et seqq.

Calling Subroutines

- Calling a subroutine is not different from calling a built-in Tcl command:
  - Syntax analysis takes place including all kinds substitutions, honouring quoting etc.
  - The subroutine's name is looked-up according to the first word and ...
  - … all other words of the command line are passed on as arguments

Some of the Tcl commands that you already have used might actually have been Tcl-coded subroutines.

Make sure you have understood that – from the viewpoint of the caller – there is no difference between

- calling any of Tcl's pre-defined commands or
- a function defined with Tcl's `proc` command.

144
et seqq.

Value Arguments

- Passing per value is the default for arguments
  - Inside the subroutine the argument behaves like a variable and may have its value modified
  - This is never reflected to the call-site, even if the argument happens to be a variable

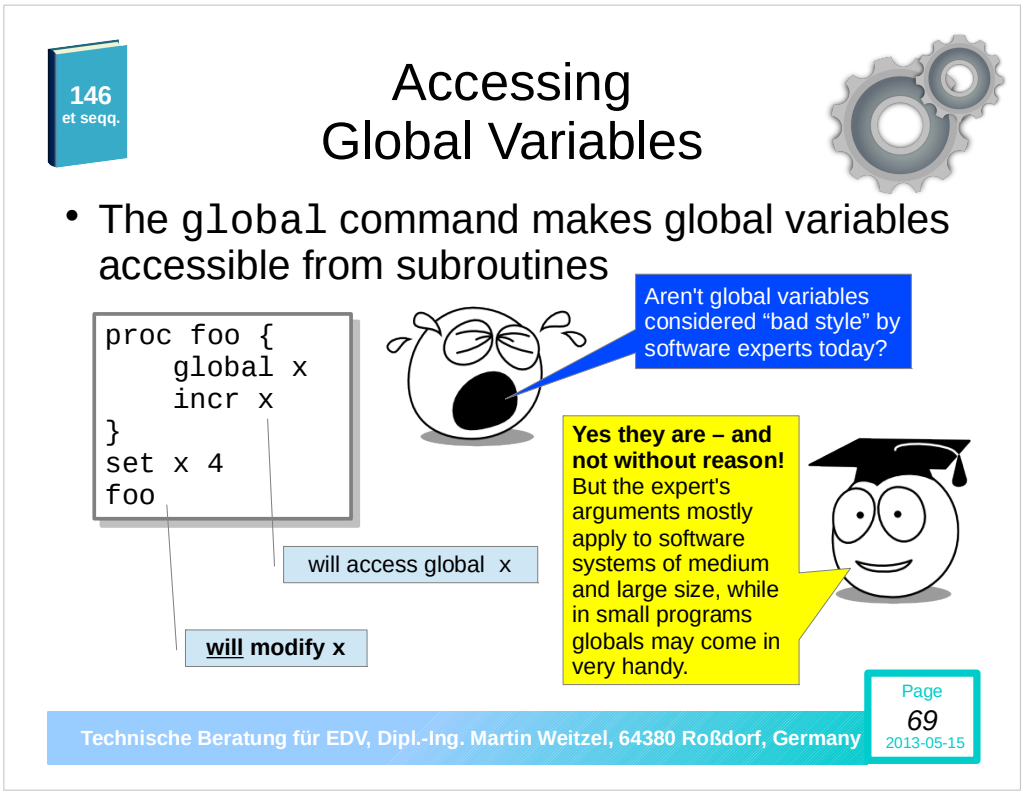
```
proc foo {arg} { …; set arg 42; … }
foo 20
set x 4
foo x
```

will modify arg internally

how could 20 be set to 42?

will not modify x

Using value arguments by default matches C/C++ style ...



The necessity to explicitly specify which global variables are used in a functions makes Tcl different from C/C++ (though at least in C++ access to a global variable may be made explicit with a prefixed scope operator “::”).

148
et seqq.

Reference Arguments

- The `upvar` command allows in a subroutine to reference variables local to the caller.

```
proc foo {} {vname} {
    upvar $vname v
    incr v
}
set x 41
set y 1000
foo x
foo y
```

will make v an **alias for a variable in the callers scope** which name is given by `$vname`

will modify the callers variable via that alias

will set x to 42

will set y to 1001

While you are not (yet) comfortable with Tcl subroutines, it's sufficient to apply this technique in “cook-book” style.

Actually reference arguments are conceptionally similar to pointers in C/C++, but its use recipe is quite different:

- in the argument list of a Tcl function an ordinary name is used (while a pointer in C/C++ must be defined as such by prefixing it with an asterisk);
- what is actually transferred is – as a string – a name from the caller's scope, which denotes an accessible variable there;
- at the start of the Tcl function, the “name” handed over via the argument is made to associate the variable from the callers scope with a local variable.

Note that what is described here works recursively too:

- the “name from the caller's scope” may there have been already associated with another name from a farther (outside) scope;
- if the local name associated with a variable from the caller's scope is handed over in a function call,
  - the callee may in turn associate it with a local name;
  - by that it is able to “reach-out” (at least) two levels from its local scope.

144
et seqq.

Return Values

- The command `return`
  - immediately leaves a subroutine,
  - handing its argument as return value back to the caller of the subroutine

```
set height 20
set width 100
puts [calcArea $height $width]
```

```
proc calcArea {h w} {
    return [expr $h*$w]
}
```

As without an explicit `return` a subroutine just returns what the last executed command had returned, the following would also have sufficed:

```
proc calcArea {h w} {
    expr $h*$w
}
```


In case of “one-liner” functions that do no more and no less nothing else but calling another function, or a built-in Tcl command, that other call may or may not be part of a command substitution and a `return` statement.

Both of the following have the same effect:

- ```
  proc foo {x} {
      return [bar $x]
  }
  ```
- ```
  proc foo {x} {
      bar $x
  }
  ```

While the above would work for the last statement of functions of any length – as always the return value of the last command executed is returned – it should be considered with caution:

- When it is essential that a function returns the same value as another one, called from it, it is usually better to make this embedded call explicit with a `return` command and a command substitution.
- Otherwise, if the function already contains several commands, it is likely that more commands will be added at the end, or the order of the contained commands might be changed, that way inadvertently disturbing the required order for returning the correct value.

Also, functions executing an `if` command as the last thing should be considered attentively, as their return value may depend on how the condition evaluates and which branch is taken!

146
et seqq.

Argument Default Values

- Default values may be supplied together with the argument list in the `proc` command

```
# increment any variable, also
# floating point, as incr only
# supports integral values

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

Returning the incremented value mimics the behaviour of `incr` in that respect.

Be sure to have understand that the second argument to Tcl's `proc` command is actually

- a list of lists,
- where each of the embedded list(s) may have exactly one or two elements:
  - the first is the formal argument name;
  - the second, if present, is the default value.

As a not too far-fetched example, consider the case where a function has too string arguments, both of which shall have the empty string as default value:

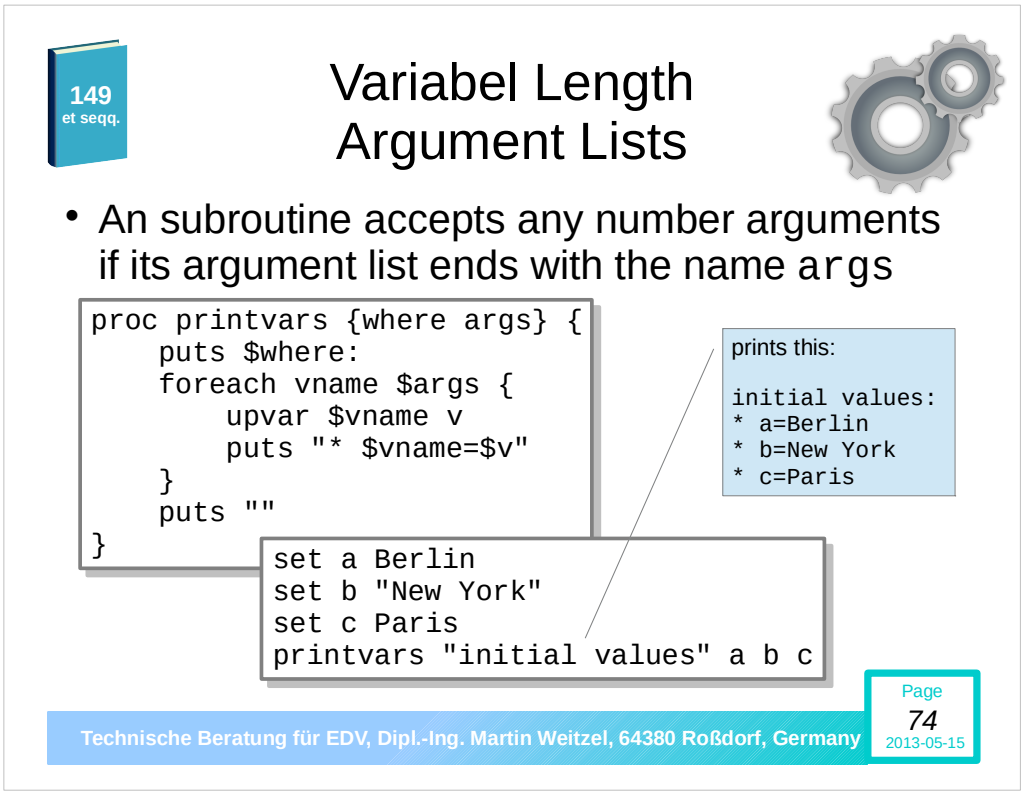
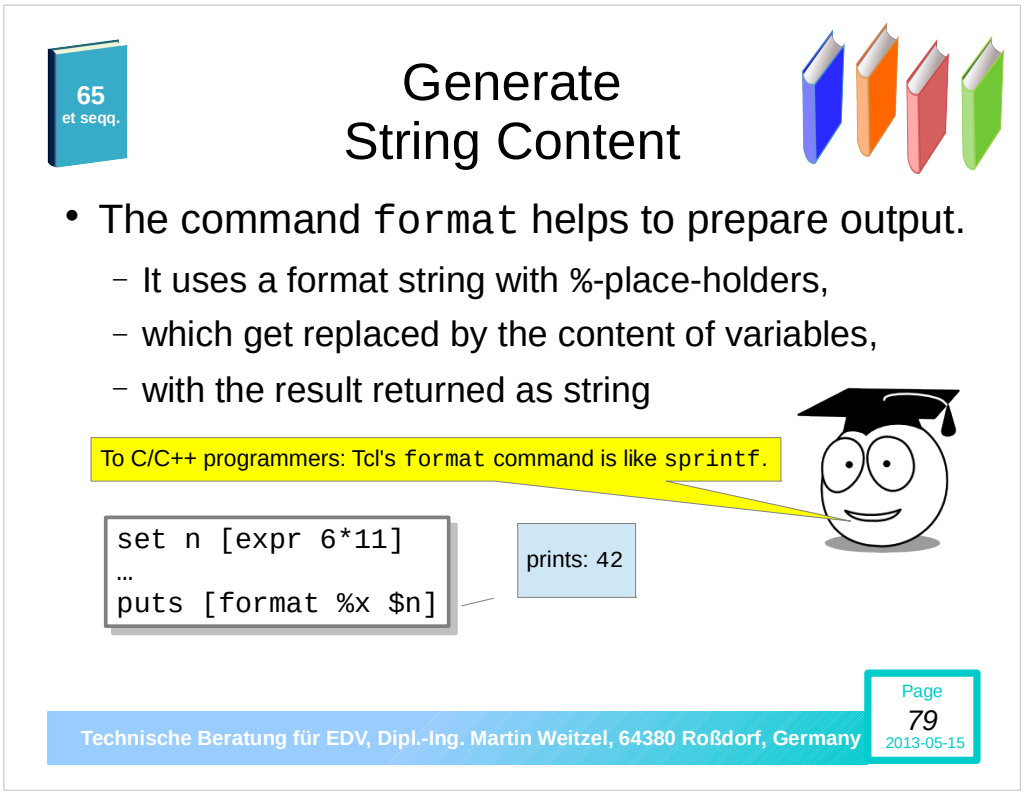
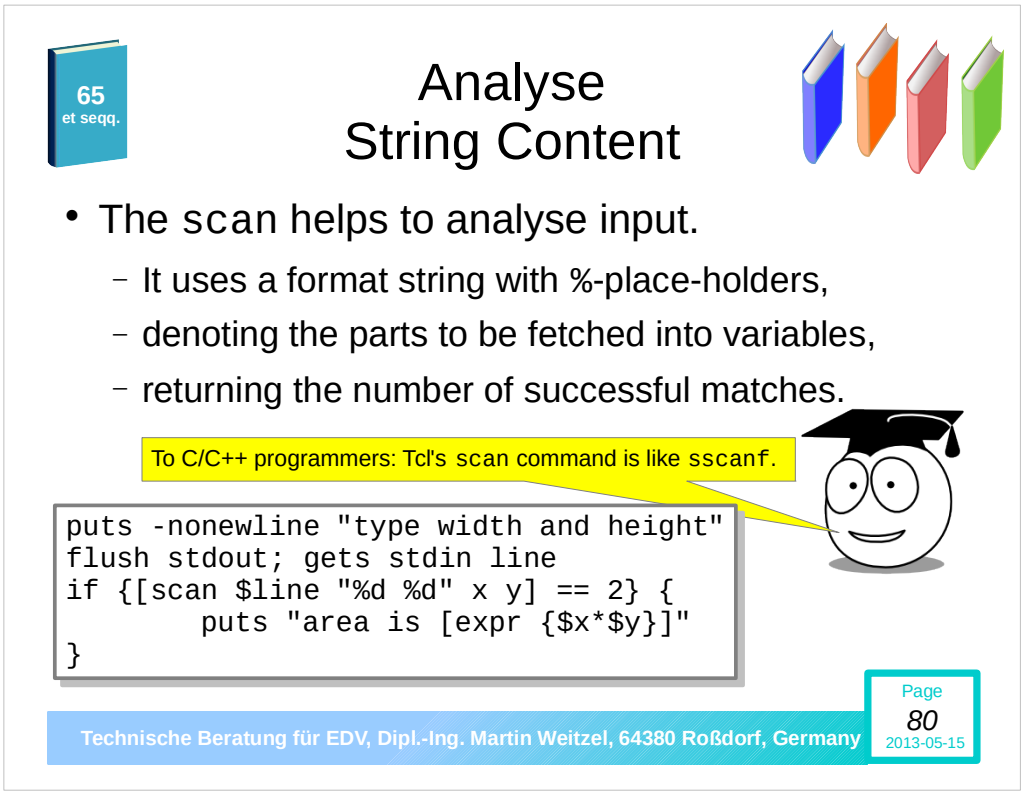
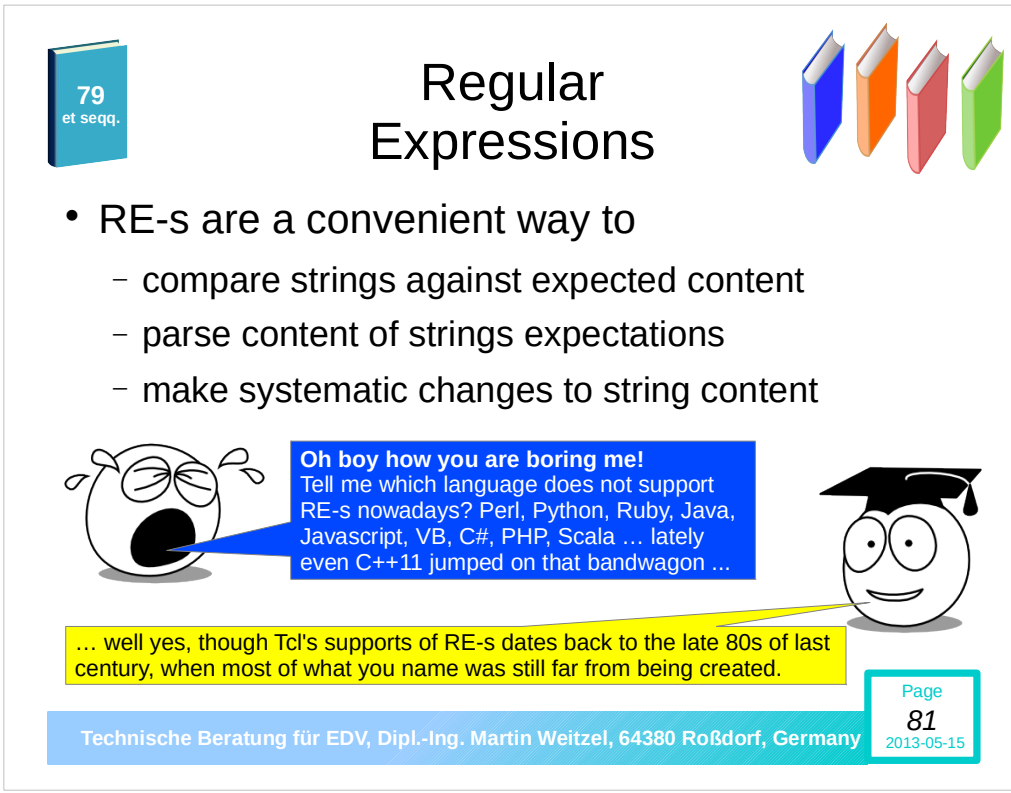
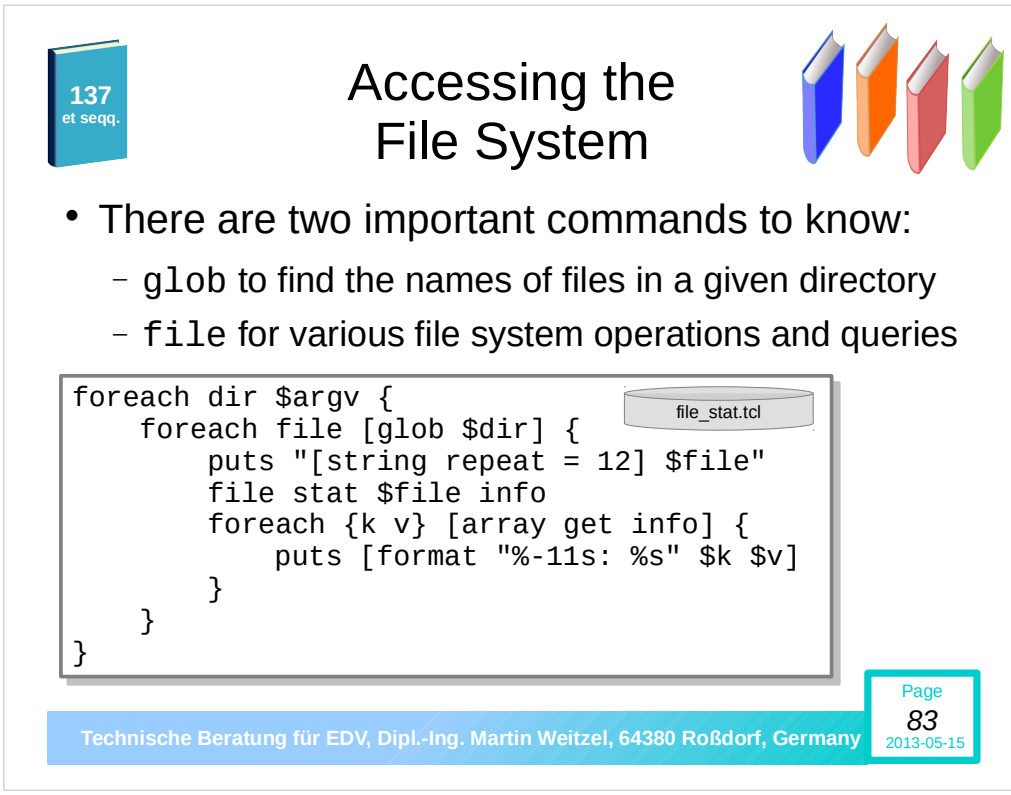
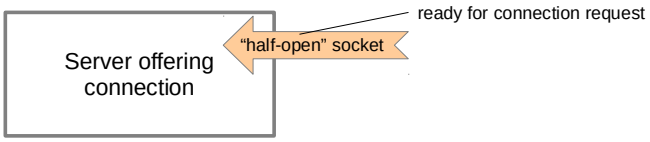
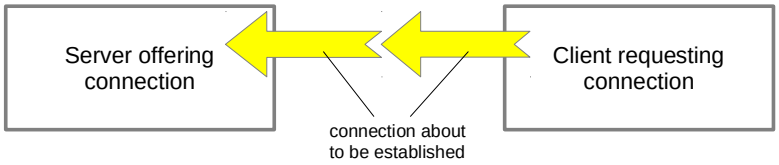
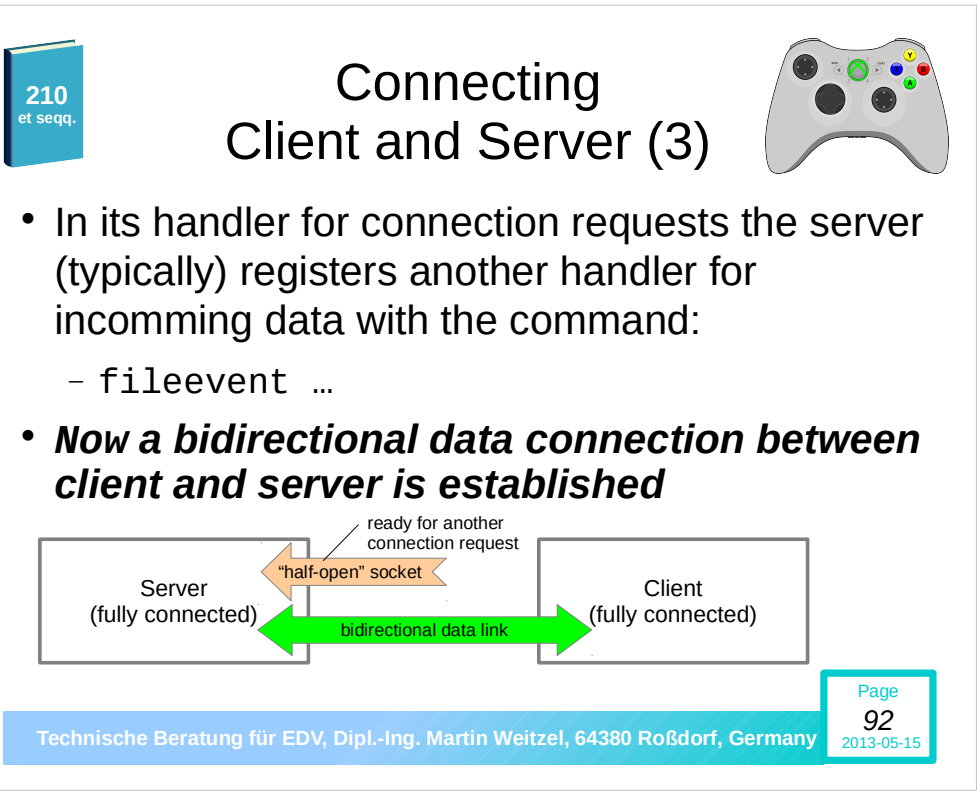
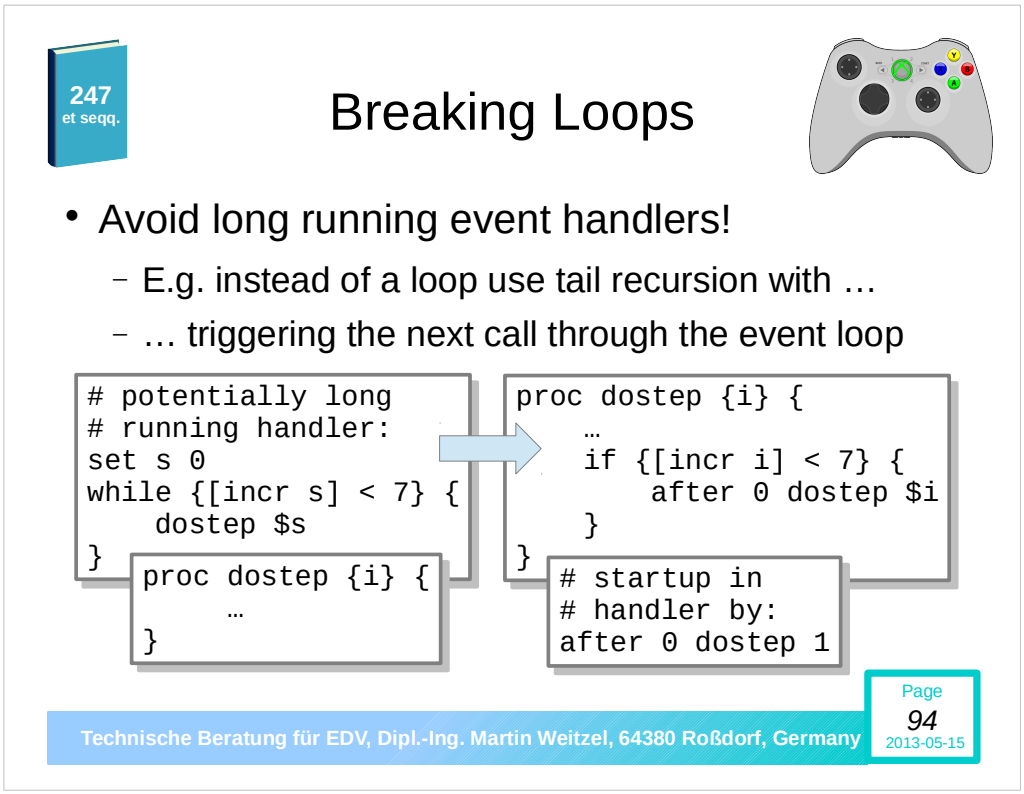
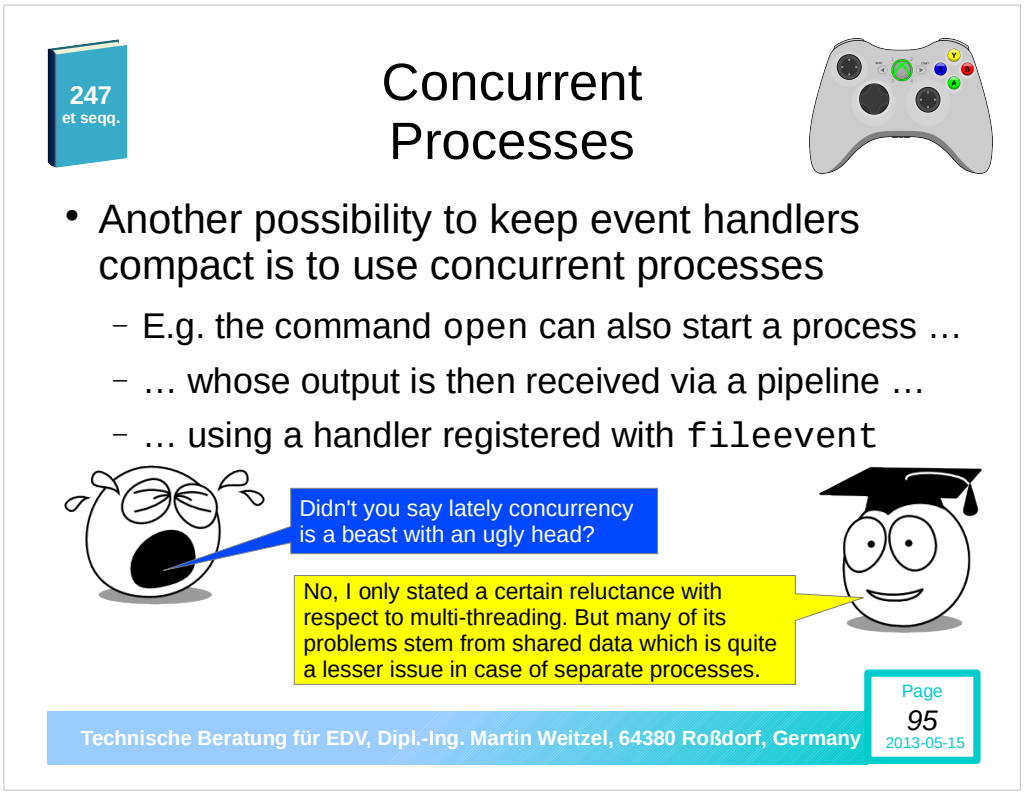
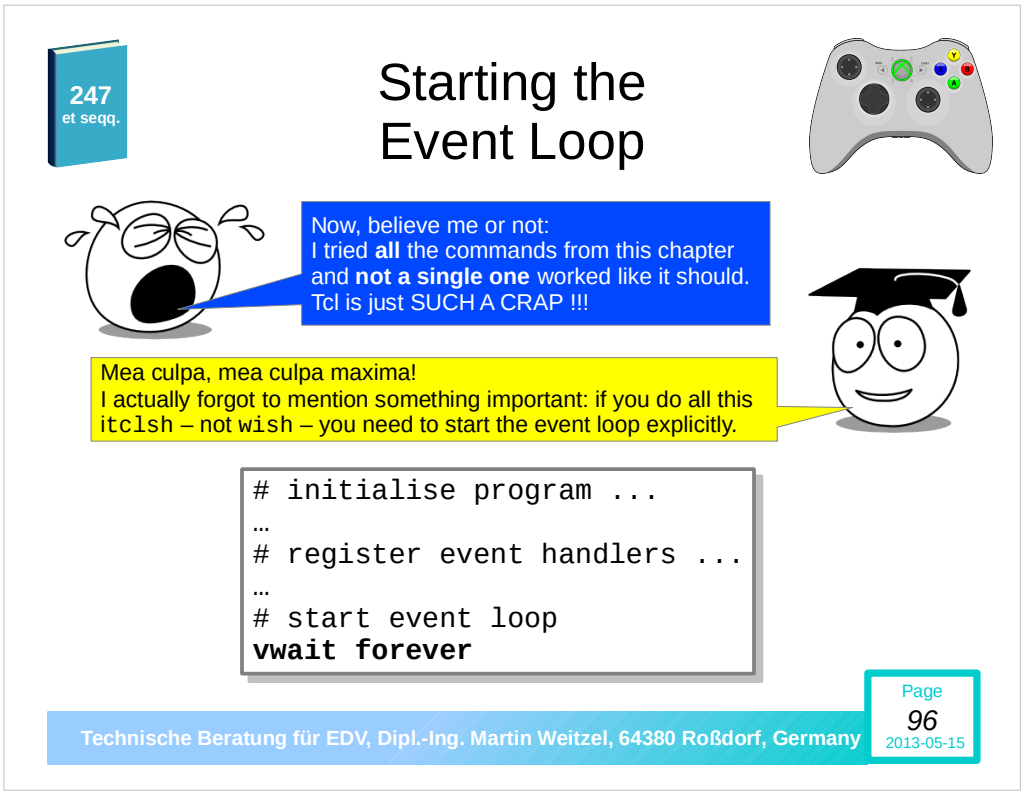
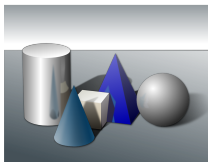
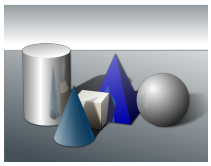
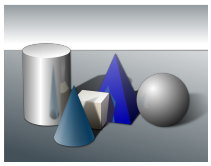
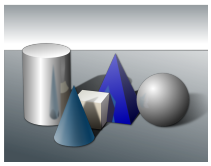
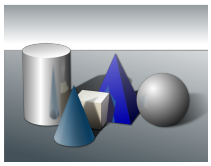
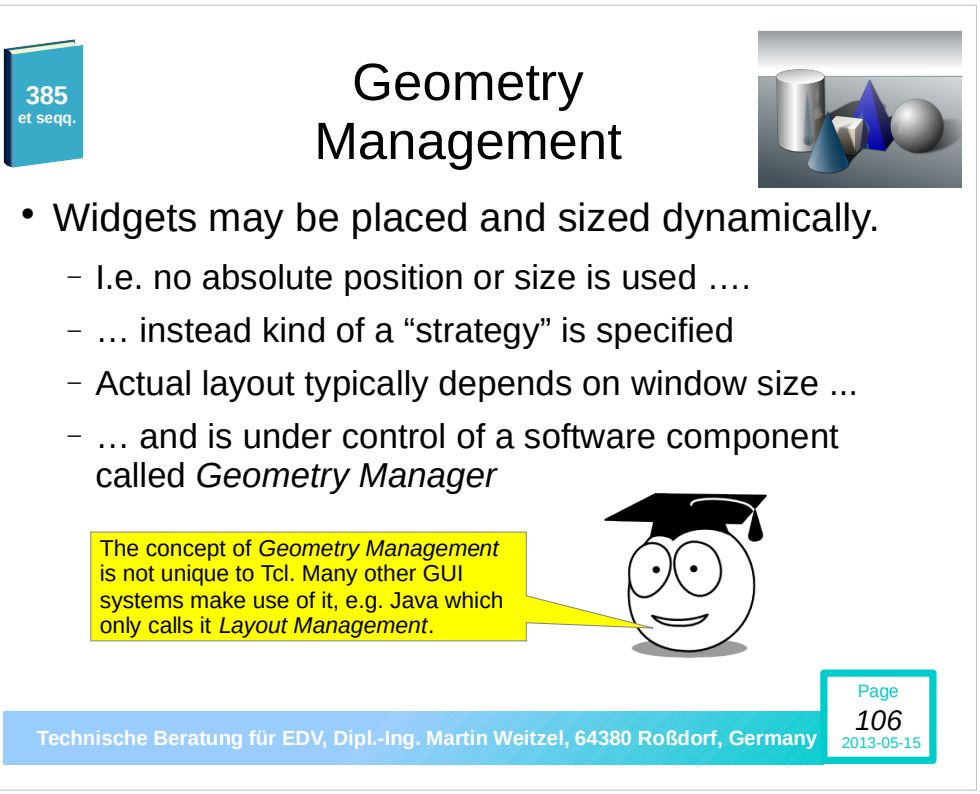
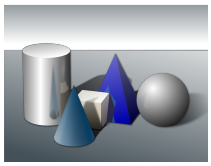
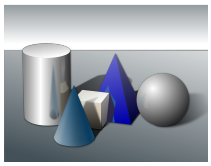
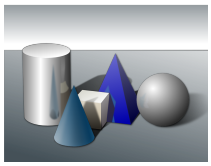
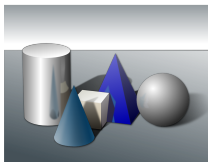
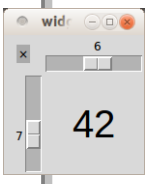
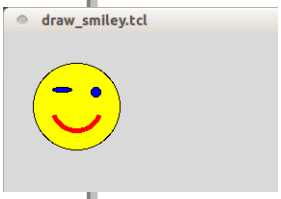
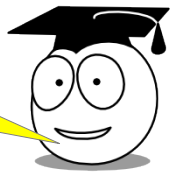
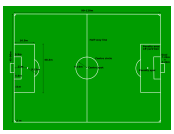
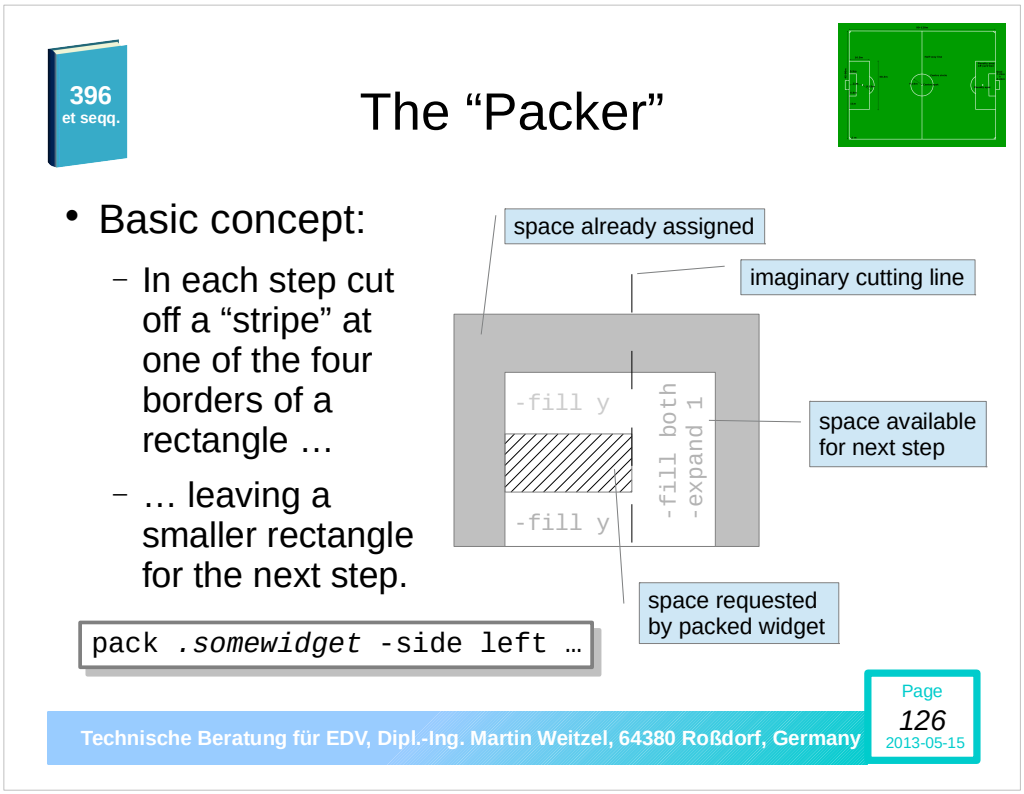
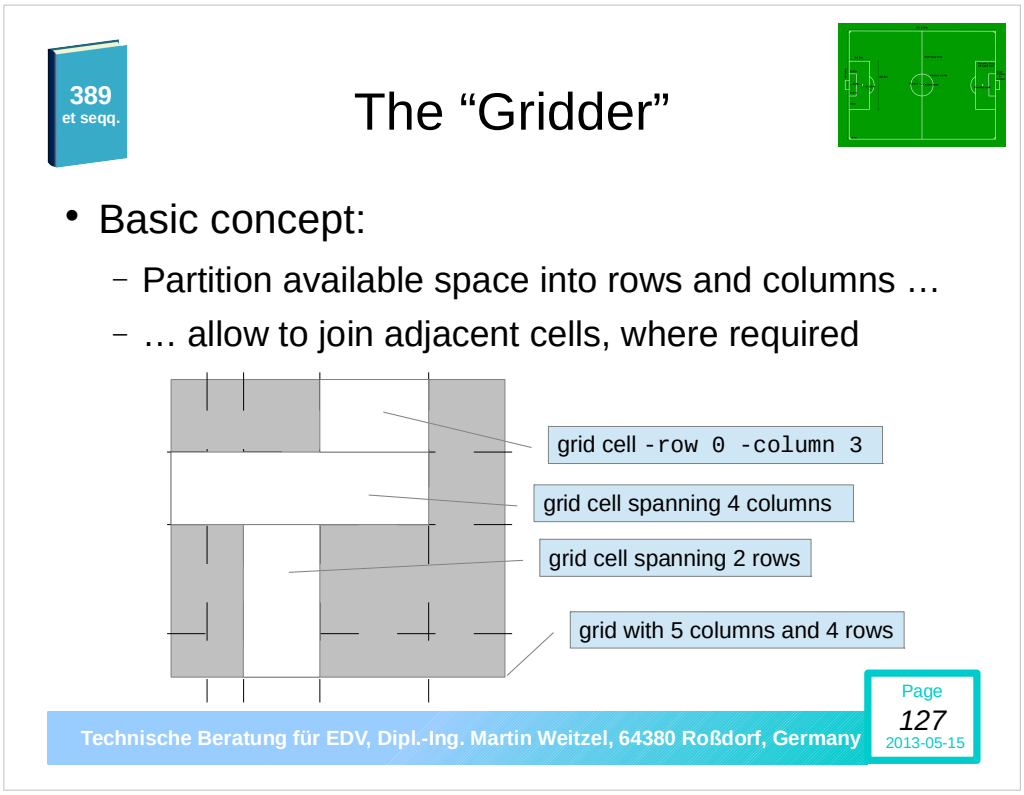
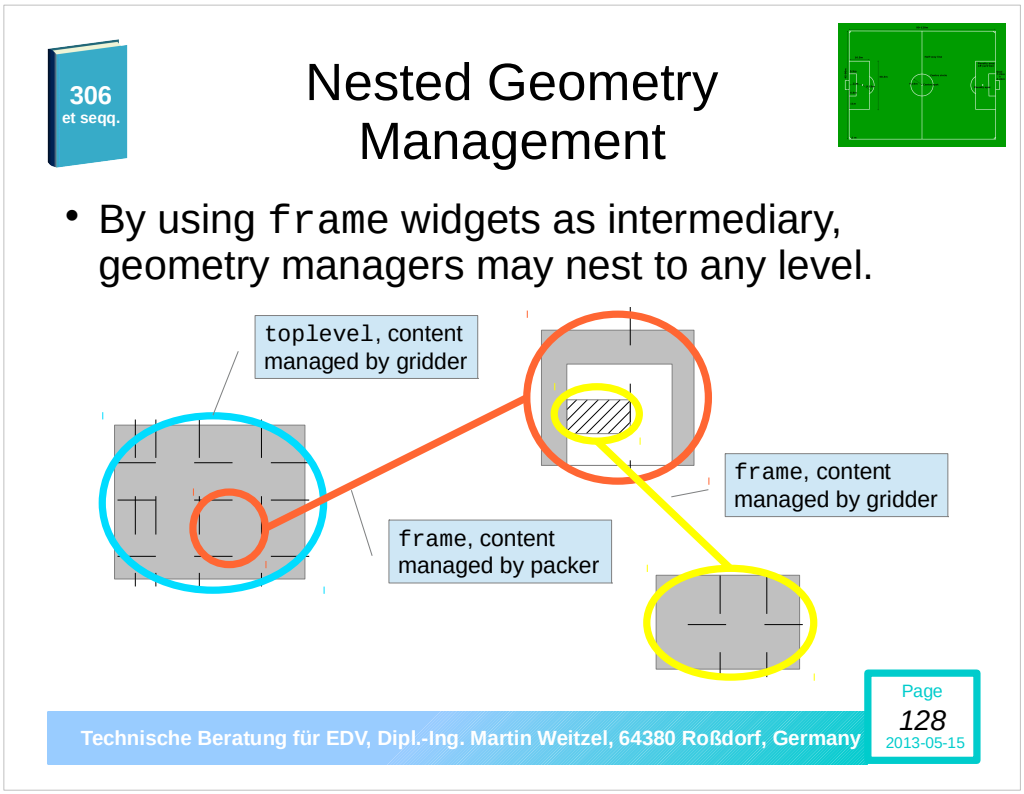
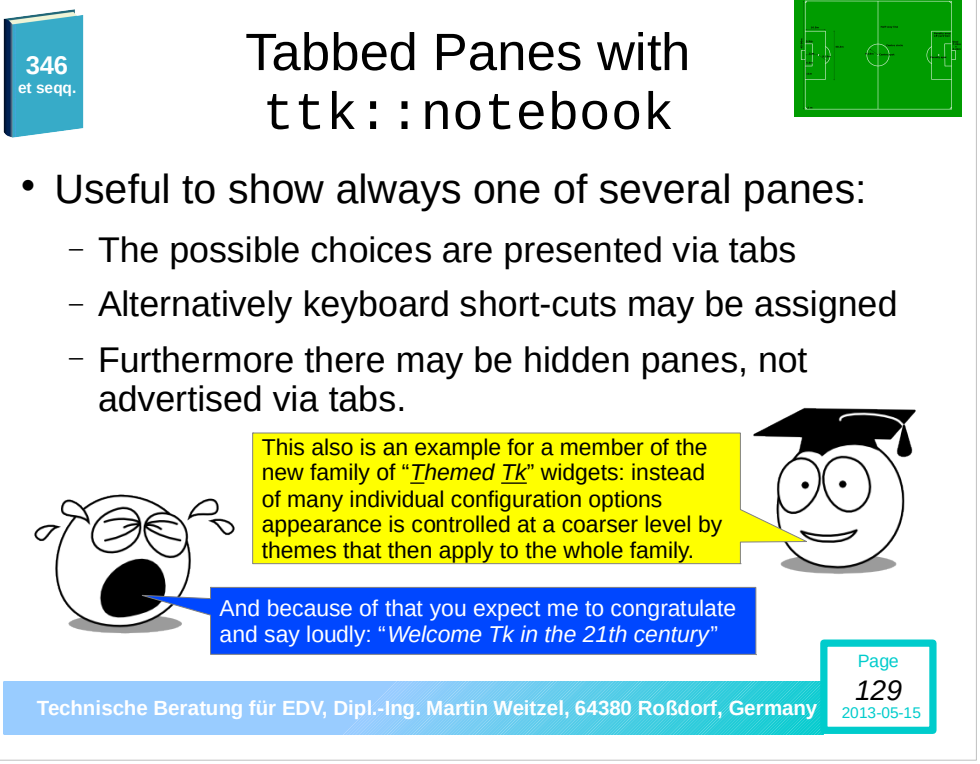
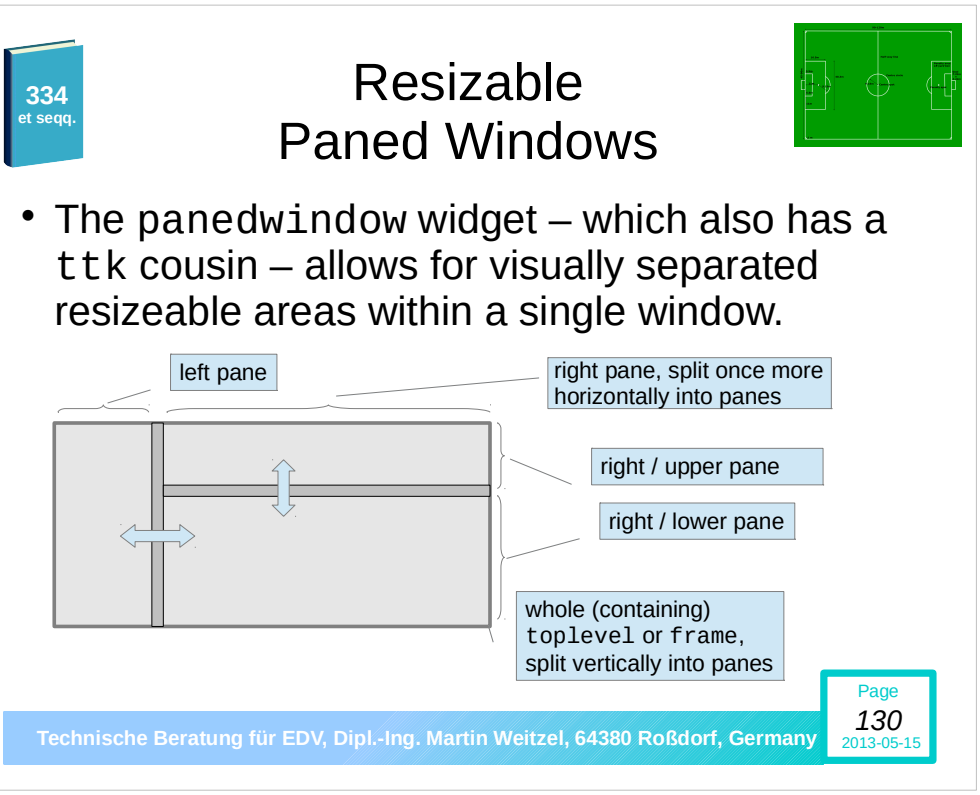
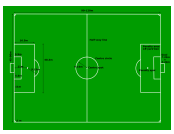
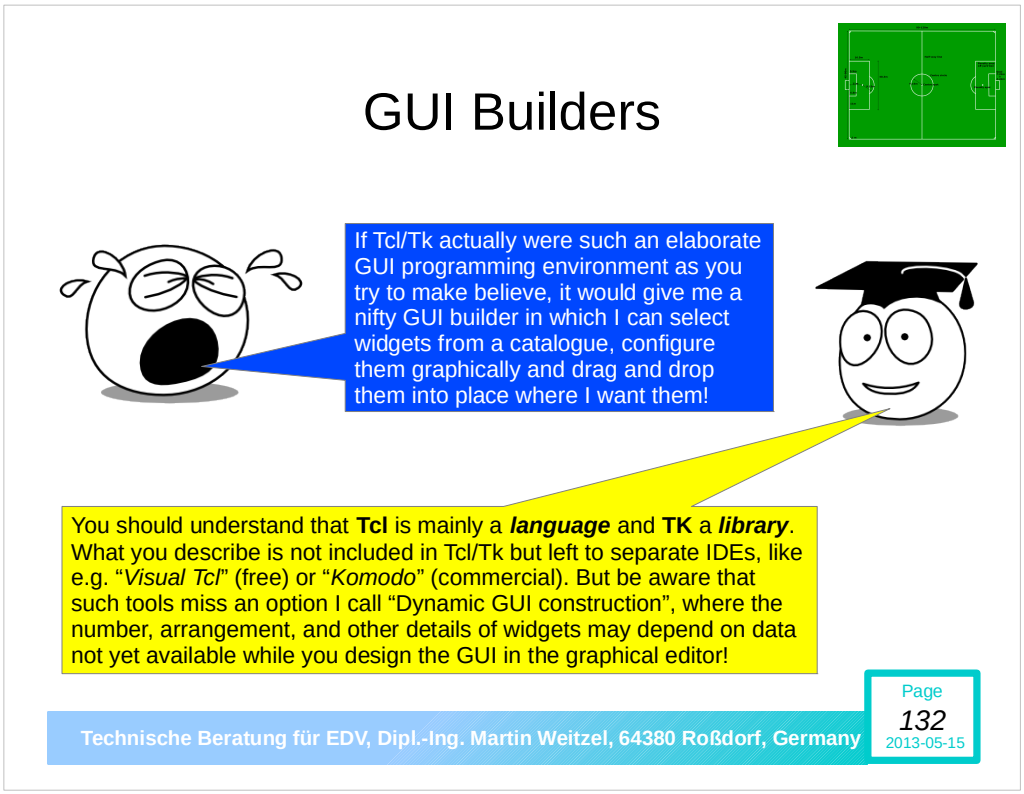
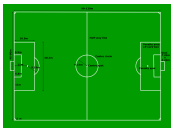
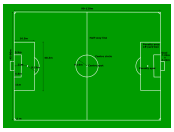
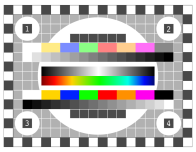
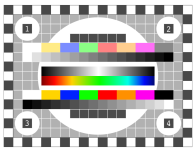
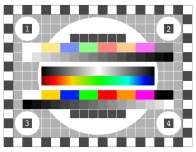
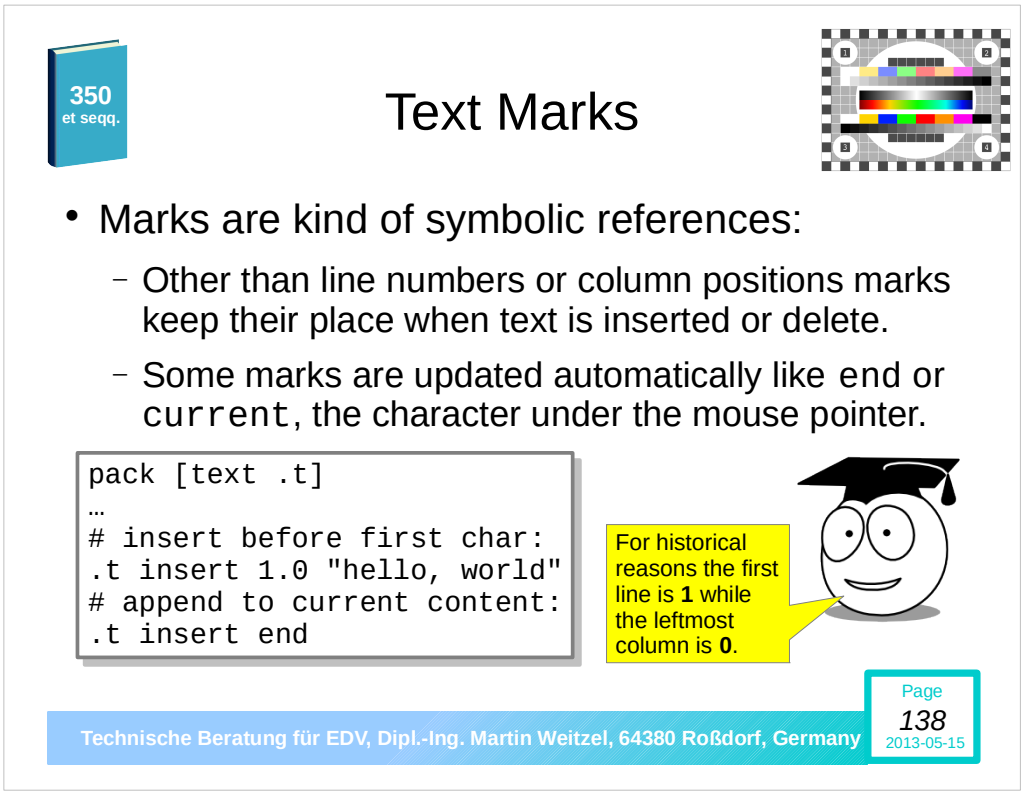
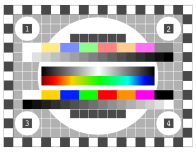
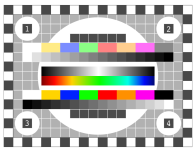
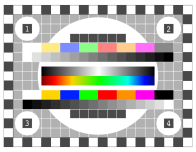
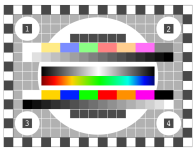
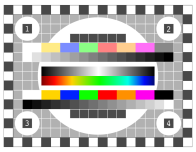
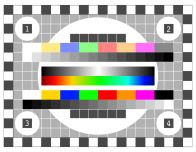
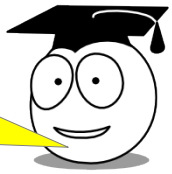
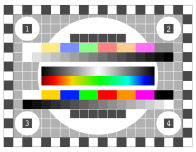
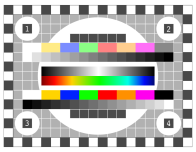
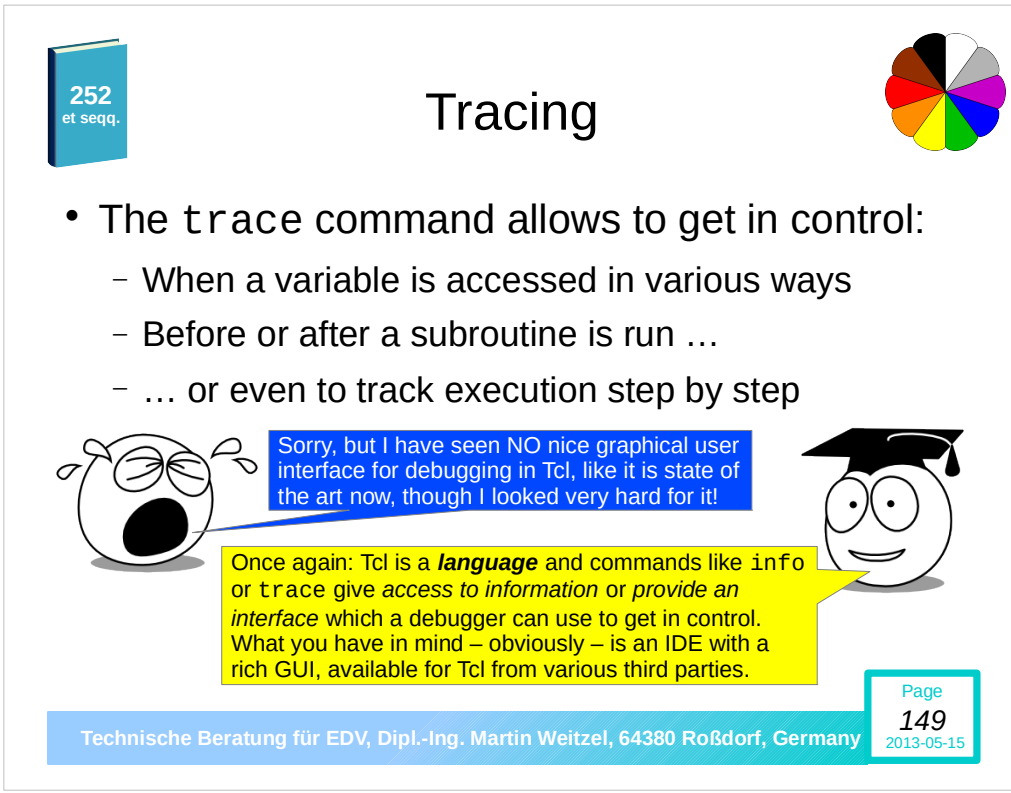
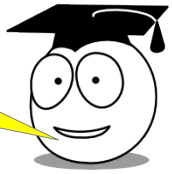
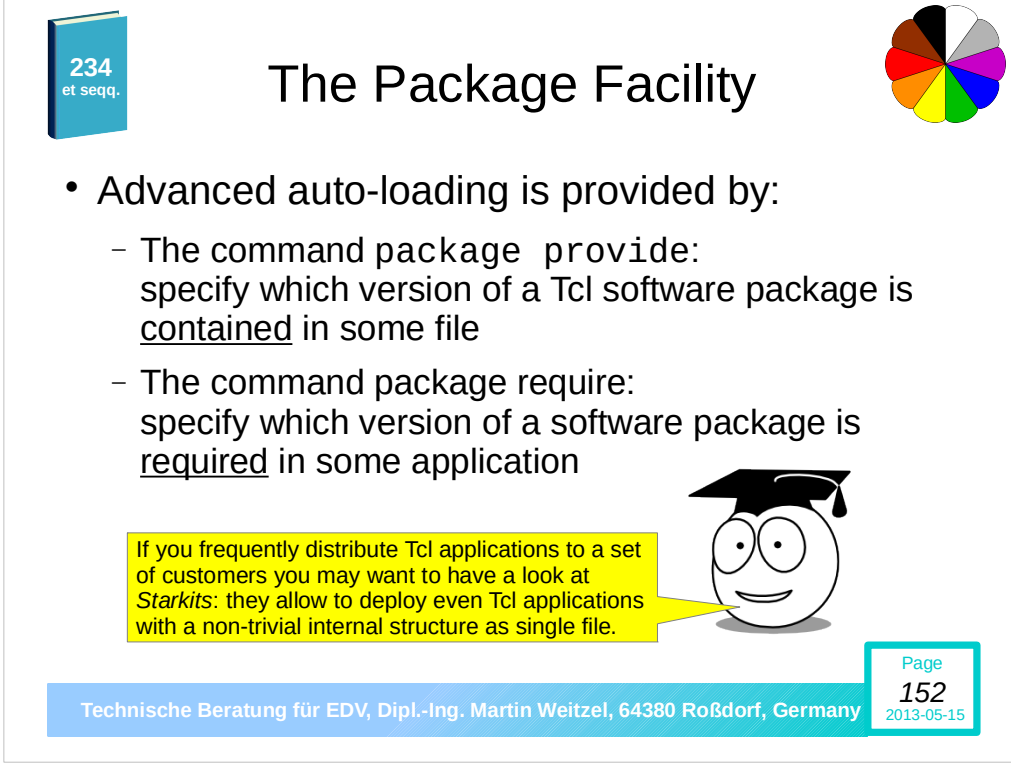
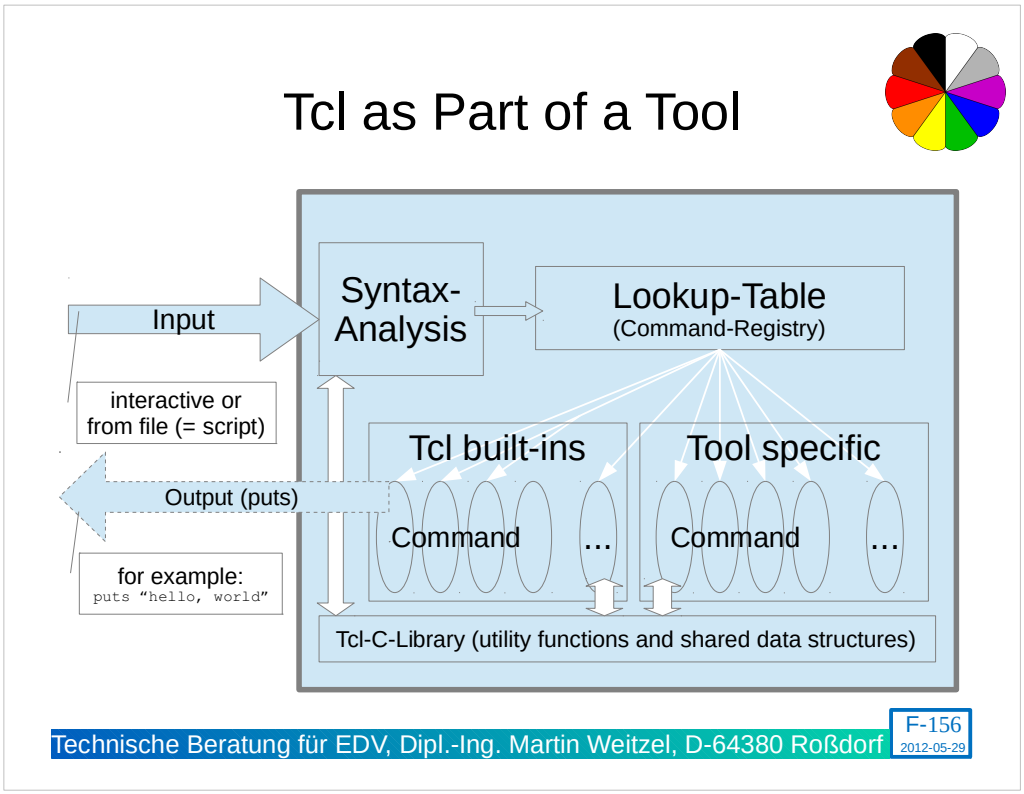
- ```
  proc foo {{a {}} {b {}}} {
      …
  }
  ```

Q: Can you explain the purpose of each of the curly braces in the argument list above?

Q: Could some brace be omitted?

Q: What if the following is used instead?

- `proc foo {{a ""} {b ""}} …`

Q: What changes if the first argument shall have no default value?
  ```

Tcl Arrays as Structure Surrogate

- Tcl arrays can also be used like struct-s:
  - A subroutine may receive or return several at once via a single array (reference)
  - It only has to agree with the caller on the meaning of array index names

```
array set r {
    height 20
    width 100
}
calcArea r
puts $r(area)
```

```
proc calcArea {an} {
    upvar $an a
    set a(area) [expr $a(height)*$a(width)]
}
```



You might need to contemplate some time over this:

- ```
  proc foo {args} {
      puts [llength $args]
  }
  ```
- ```
  set li {a b c}
  foo a b c ;# output is: 3
  foo $li   ;# output is: 1
  ```

Then you may (or may not) understand this:

- ```
  proc foo {args} {
      puts $args
  }
  ```
- ```
  set li {a b c}
  foo a b c ;# output is: a b c
  foo $li   ;# output is: {a b c}
  puts $li  ;# output is: a b c
  ```

Time for some Practical Exercises

1. Stepwise analyse and compare the various programs (`param_demo1.tcl` to `param_demo7.tcl`) demonstrating different ways to pass in and out information to and from subroutines.
2. Compare the effects of the various programs (`error_demo1.tcl` to `error_demo3.tcl`) demonstrating some error handling.

- Optional: fill in the “TO-DO”-s in the series of programs `x_mathfuncs1.tcl` to `x_mathfuncs5.tcl` (and/or compare these with the solutions in `mathfuncs1.tcl` to `mathfuncs5.tcl`).

Though parts 1 and 2 of this exercise only require to analyse and compare various versions of a demo program, afterwards you should be able to answer the following questions:

- In `param_demo1.tcl`, why are there two `global` commands?
  - Could these be avoided altogether?
  - Could these be merged into one or split into four `global` commands?
- In `param_demo2.tcl`, what is the return value of the `prtable` function?
- In `param_demo3.tcl`, which of the arguments have default values?
- In `param_demo4.tcl`, in the call to `prtable`, why are there curly braces around the set of values to be used as *start*, *end*, *step*, and the list of math-functions?
- In `param_demo5.tcl`, in the call to `prtable`, why are there **no** curly braces around the set of values to be used as *start*, *end*, *step*, and the list of math-functions?
- In `param_demo6.tcl`, what is the `upvar` command good for in the function `prtable`?
- In `param_demo7.tcl`, what happens if `prtable` is called with no arguments?
- In `error_demo1.tcl`, why does the first call to `foo` work OK but the second crash the program?
- In `error_demo2.tcl`, which change prevents the program crash here?
- In `error_demo3.tcl`, why need there be square brackets around the `catch` command in `foo` but no curly braces, as the usually surround a

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

With respect to the final result of the `mathfuncs`-series:

- Do you think the invested effort is worth the result?
- Or is this rather an example for over-engineering and would you have stopped earlier?
- If so, with which of the prior versions?

Part 5
Library Functions

- What's ahead?
  - String handling
  - Regular expressions
  - Working with files

Using library functions avoids to reinvent the wheel over and over again.

Moreover, library functions are usually well tested and may use optimization techniques of proven experts.

Finally library functions are usually well documented, other than many home-brewed and ad-hoc solutions.

General
String Processing

- The `string` command provides facilities to
  - determine string length
  - string comparisons
  - access parts of strings
  - modify strings
  - … (and more) ...



You may easily gain experience with Tcl's `format` command by trying small examples interactively (in a `tclsh` or `tkcon`).



As with `sscanf` and `fscanf` in C, when using Tcl's `scan` command it is a bit tricky to detect “extra characters” after what is expected.

In the above example

- the input `12 15` leaves `12` in `x` and `15` in `y`,
- but the input `12 15abc` does this also,
- and `12 150` too, infamously … (you actually spotted the `0` that wasn't a `0`?)

An easy technique to avoid thos builds on the following idea:

- The newline character removed by Tcl's `gets` command is again added,
- explicitly tested for with the `%[\n]` place-holder in the `scan` command,
- and – of course – included in the expected match-count.

Putting these three things together results in the following, improved recognition technique (modifications to the original example marked in bold):

- `if {[scan $line\n {%d %d %[\n]} x y] == 3} …`

Note that the format specification is put into curly braces above, otherwise the opening square bracket will have to be quoted so that it doesn't trigger command substitution.

Q: If instead of curly braces the format specification is put into double quotes, will it then also be necessary to double the backslash in `\n`?



Maybe you know regular expressions already from some other programming or command language.

Otherwise consider to get some training with an *Interactive Regular Expression Tester*. There are many now on the internet, from simple to sophisticated.

Two rather simple ones are included with the examples:

- `regexdemo.tcl`
- `rematch.tcl`

185
et seqq.

Working with Files

- File access in Tcl is modelled much like in C:
  - The command `open` establishes the connection
  - It returns a *channel* to be used in commands working with the file contents, like `gets`, `puts`, `flush`, `seek`, `tell` ...
  - Finally the command `close` releases the connection

Though you usually store the channel returned from `open` in a variable and hence could look at it (e.g. print it with `puts`), you usually won't: its content is not really interesting – much like the content of the `FILE` structure to which a pointer is returned by the C library function `fopen`.

If you have some experience how to work with files in C, the basic approach can be easily adapted to Tcl. Following are three simple examples (omitting decent error handling):

```
# list content of file, prepending line numbers:
set file [open [lindex $argv 0] r]
set line_nr 0
while {[gets $file line] >= 0} {
    puts [format {%4d %s\n} [incr line_nr] $line]
}
close $file
```

```
# compare two files bytewise:
set f1 [open [lindex $argv 0] r]
set f2 [open [lindex $argv 1] r]
set b 0L ;# maybe large files …
while {[read $f1 1] == 1 && [read $f2 1] == 1}
{incr b}
if {![eof $f1] || ![eof $f2]} {
    puts "first difference at offset $b"
}
close $f1; close $f2
```

```
# copy one file to another one:
set infile [open [lindex $argv 0] r]
set outfile [open [lindex $argv 1] w]
set content [read $infile]
puts -nonewline $outfile $content
close $infile; close $outfile
```



The Tcl command file allows a number of manipulations of files as a whole.

Besides many other possibilities it allows to copy the content of one file into another. So it can provide an alternative solution to what the last example from page 84 does.

Here we go (improved with error checking this time):

```
# copy content of file:
if {[$argc != 2]} {
    puts stderr "Usage: $argv0 from_file to_file"
    exit 1
}
set from_file [lindex $argv 0]
if {![file exists $from_file]} {
    puts stderr "$argv0: $from_file does not exist"
    exit 2
}
set to_file [lindex $argv 1]
if {[file exists $to_file]} {
    puts stderr "$argv0: $to_file already exist"
    exit 3
}
if {[string equal $from_file $to_file]} {
    puts stderr "$argv0: can't copy $from_file to
itself"
    exit 4
}
file copy -- $from_file $to_file
exit 0
```

Time for some Practical Exercises

- Adapt the program from page 83 so that
  - it recursively descends into sub-directories,
  - does not list any information about the files,
  - except the last component of its path name,
  - using appropriate indentation to indicate the structure of sub-directories.

Optional Practical Exercises

- If you rather want an exercise for regular expressions, fill in the “TO-DO”-s in the series of programs
  - `x_regexp1.tcl`
  - `x_regexp2.tcl`
  - `x_regexp3.tcl`

If you want to learn rather from the solutions, these are available here:

- `regexp1.tcl`
- `regexp2.tcl`
- `regexp3.tcl`

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Which of the text pattern recognitions in the various sections of `regex1.cpp` …

- … might be more easily achieved with string processing (or other library) functions?
- … would be much harder to carry out without Tcl's regular expressions?

Part 6
Event-Driven Design

- What's ahead
  - Asynchronous I/O
  - TCP/IP sockets
  - Timed Events

Tcl strongly favours the *Even-Driven*-Approach in situations, where otherwise frequently *Mutlti-Threading* springs into mind first.

While there is a clear advantage of multi-threading when the performance of an application should scale with the number of CPU cores, an event driven design

- behaves more deterministically,
- is better testable, because of same output for same input
  (as long as the sequence of events is reproducible)
- will neither deadlock, nor show race conditions

and hence is – compared tu multi-threading – much less error prone

On the other hand, a language or framework must explicitly support the event driven style (what Tcl does, as this chapter will show) and in addition the developer needs to understand a small set of rules, i.e. especially avoid the “No-Go”-s (which also are the topic of this chapter).

Generally the event-driven approach centrers around

- an event loop and
- a number of event handlers.

It is essential that the latter are all “short-running”, which means an event-handler should never (ever) do time-consuming things, especially it should not (have to) wait for an answer of a different system component which might arrive with some delay.

206
et seqq.

Asynchronous I/O

- The command `fileevent` is important here:
  - When interfacing with the real world, data often arrives at unpredictable time points
  - Instead of “polling” for input register call-backs on file descriptors
  - This leads to elegant and efficient solutions when reading from serial ports or TCP/IP sockets
  - Callbacks may also be registered for output, to avoid being blocked by temporary buffer fill-ups

To properly support event driven application design, any input or output in Tcl can (optionally) be processed by call-backs, registered with the `fileevent` command.

- This is usually done for input, as it is essential here when the data source delivers in chunks and/or at unknown time
- For output it is less frequently necessary, as the default buffering of many data sinks is typically sufficient. But t make an application really robust it might need to be prepared for situations where some output buffer becomes full and further (synchronous) output would block inside the event handler – a thing absolutely to avoid if an event driven designs should keep its usual responsiveness.

TCP/IP Sockets

- There are two callbacks for TCP/IP socket
  - First while waiting for incoming connection requests (register callback with the `socket` command)
  - Second when waiting for data (register callback with the `fileevent` command)

Compared to C/C++ programming using TCP/IP sockets in Tcl is really easy and fun: a simplistic web-server – to which you can connect with any ordinary browers – can be done in less than a dozen lines of code.

Be sure to have understood that **none of the above commands actually causes a delay** (of indeterminable length), they just register a function that will be called later.

Connecting Client and Server (1)

- The server (starts and) – at some point – executes the command:
  - `socket -server …`
- ***Now there is a handler registered which will be called in response to incomming connection requests***



TCP/IP-sockets are frequently used in client-server applications as data connection.

Being bidirectional by nature it can sometimes be difficult to decide the roles – which side is the server and which the client, though

- usually the side that initiates the connection is called “client” and
- the side that waits for clients to connect is called “server”

Connecting Client and Server (2)

- The client – at some point – executes the command:
  - `socket` … (i.e. without the `-server` option)
- ***In the server, this causes the execution of the handler registered for incoming connection requests***



Once it is in the handler for incoming connection requests, the server cannot process any other events. **Hence nothing should be done that might cause delays of indeterminable length.**

What is usually OK (and may be sufficient in simple cases) is that the server sends a direct response to the client and immediately closes connection. (With this style of "one-shot" communication there is no need to register another handler in the server with `fileevent`.)



Defining the roles of client and server as has been done here makes it irrelevant which side is the first one to "talk" to the other, once the connection is established. In fact it may be any of both.

But of course there needs to be some protocol defining the message exchange in detail, including behaviour in case of protocol violations and time-outs.

The role of the communication protocol is crucial for a robust client-server architecture:

- No side should rely on bug-free implementation of the protocol on the other side.
- Then, to some degree, errors in client-server communication may be "self-healing"

Also, depending on network protection, both sides might need to take the possibility of malevolent connection attempts into account.

This already effects one-shot connections (see last page): An intruder who is able to put himself into the position of a (malevolent) server might then hang a client by accepting the connection and simply leaving the socket open, without ever sending anything.

Timed Events

- Sleeping in an event handler is a clear No-Go!
  - Instead send a timed event to be handler after the required delay
  - This can be done with the `after` command

Methinks all this event stuff abracadabra was only put into Tcl as a lame excuse for not having to implement true multi-threading, as most any other programming language supports it now.

And methinks **I love event-driven designs** – maybe because I was bitten more than once by the beasts showing up their ugly heads if you actually try to steer your ship through the wild waters of multi-threading ...

It is frequently necessary in software that some action has to be delayed by some period of time or to happen at a certain point in time – but not right now. If there is nothing else to do it then might simply *sleep* for or some time or until that point in time, but:

- Sleeping in an event handler is another **Big No-Go** for event driven designs.
- Responsiveness could not any longer be guaranteed, if some handler simply delayed.

The solution are *Timed Events*, which in Tcl an application can send to itself by means of the `after` command.



Less obviously the Tcl `after` command also offers an elegant solution if the work, an event handler has to do, is long running by its very nature.

- Most typically long-running tasks are not long, linear sequences but involve some kind of loop, or maybe even nested loops.
- The trick is this:
  - Instead of running a loop inside an event handler …
  - … run only its body once and send an event with a delay of 0.

That's it!

If a language or framework supports the event-driven approach, it usually has an event-queue to store events coming in while a handler is running, so also Tcl has one. Therefore, if repeated runs through a loop are achieved by an event-handler that triggers its own (re-) execution at its end, any other events that came in while the handler was running will have been stored in the queue.

So, event-handling never stops completely but intermingles with the long running task.

In addition it is very easy – often easier or at least more natural as in case of multiple threads – to stop the long running task prematurely: if some other event, arriving while the task still runs, causes it completion of that task to be pointless, it can simply set a (global) flag to be checked each time the next pass starts through the code block which originally constituted the body of the loop.



In case it seems to difficult to break some long-running task into pieces – maybe it calls some C/C++ library function with a loop that cannot be broken – a second option is to run this in a separate process.

The asynchronous part then needs only to care for starting that process and then

- supply it with data,
- (maybe in chunks, as necessary),
- retrieve results
- (and maybe intermediate results or some kind of progress indicator).

All of this can easily be handled with socket or pipeline communication and handlers registered with Tcl's `fileevent` command.



When a Tk GUI is run (i.e. in `wish`) the event loop runs automatically, as it is required to receive (at least) keyboard, mouse, and window events.

Without Tk, i.e. when only Tcl features are of interest and therefore running `tclsh` is sufficient, the event loop must be explicitly run with Tcl's `vwait` command.

Otherwise an application may well register handlers or send delayed events to itself, but the associated handlers will never be run!

Ending the Event Loop

- The event loop started with `vwait` runs until
  - Some handler executes the exit command, or
  - The variable named as argument is assigned to
- The event loop will typically NOT end
  - If a handler throws an error …
  - … then the `bgerror` procedure is called …
  - … which per default does not call exit

Often in an event-driven Tcl application the event handling loop never comes to an explicit end – instead the whole application is terminated in an event-handler that executes the `exit` command.

If some central shut-down code is to be run at the end of the program, it might be run after the event-loop has terminated, so that the overall structure looks symmetrical:

- Run start-up code (includes registering first event handlers)
- Enter event-loop
- Run shut-down code

Alternatively the shut-down code may be put into the same event-handler that (finally) executes the `exit` command, or in a subroutine called from that handler.

Time for some Practical Exercises

- Review and run the example `socket-demo.html`:
  - as far as possible explain the output and
  - note your open questions for the discussion
- Review and run the example `clock-demo.tcl`:
  - try to understand how the slider controls the speed of the clock and how it can run even backwards
  - make it run from 19:59 to 20:01 …

Event-driven designs have many applications, though in Tcl they are usually found when at least one of the following condition holds:

- The application has a more or less complex client/server architecture – frequently but not necessarily based on TCP/IP communication.
- The application has a GUI – i.e. it's actually the Tk part of Tcl that is in control

Of course, both may be combined, i.e. a Tk applications may very well be (part of) a client/server architecture and not all components of it needs to be written in Tcl.

Optional Practical Exercises

- Review and run the various Client/Server examples
  - some basic instructions how to run the programs are in the comments at their beginning
  - in general you may want to use at least two shell windows: one to start the server and the other one to run the client from

This series of demo programs may be of interest (only) if you plan to implement client-server architectures with Tcl.

Feel free to experiment, like

- running several clients in parallel, or
- start the server more than once.

(If there is a local network available, you may also try to communicate across hosts, e.g. you may run a server and your colleague starts a client and vice versa.)

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Did you notice `clock-demo.tcl` displays `19:60` when it's actually `20:00` (8pm)?

Part 7
Tk Basics



- What's ahead?
  - Implicit event loop
  - Creating and configuring widgets
  - Principles of geometry management
  - Interface to window manager
  - Event bindings

Tk is introduced here in to parts (plus the small overview in the prelude).

This chapter covers the basics.

- You will then able to add a simple GUI to any of your Tcl applications.
- But adding a GUI does not mean your application cannot any longer be scripted!

Interpreter `tclsh` vs. Interpreter `wish`



- There are two differences executing a script:
  - `tclsh` reads input lines in a loop until EOF or an `exit` command and evaluating each one
  - in addition, `wish` opens a graphic window to display widgets and starts the even loop at EOF
- When used interactively
  - wish executes the event loop concurrently to reading commands, and
  - both terminate on EOF or an `exit` command

Only as a reminder: Tk can also be started from `tclsh` with the following command:

- `package require Tk`

This pops up the GUI (look out for it, some window managers choose to make it rather tiny and may put it in remote locations of a large screen) but you still can type commands to your tclsh. If such commands effect the GUI window, you will see their effect immediately promptly.

Learning by Doing



Interactively using `wish` is a great way to learn Tk, especially about its widgets and layout managers.

```
button .b -text "PRESS ME"
pack .b
.b configure -bg green
pack forget .b
pack .b -side top -fill x
pack forget .b
pack .b -side left -fill y
pack forget .b
pack .b -side left -fill both
```

at this point enlarge the graphics window by dragging one of its corners with the mouse

Be sure to understand that newly created widgets become visible only after they have been made known to a *Layout Manager*, as was done in the example above with the Tk's `pack` command.

294
et seqq.

Creating Widgets



- Creation commands are named after widgets.
  - The first argument is always the widget's name
  - It can be freely chosen as long as begins with a dot (`.`) followed by a lower case character, digit or underscore
  - Configuration options may be specified at this point …
  - … or added / changed later (see next page)

```
scale .simulationSpeed -orient vertical\
       -label "Speed (% of real time)"\
       -from 25 -to 400 -tickinterval 25\
       -variable simulation_speed\
       -command setSimulationSpeed
```


As the return value of any widget creating command is the name of the widget (always the first argument, before any option), widget creation and `pack`-ing may be combined:

```
pack [
    scale .simulationSpeed -orient vertical\
           -label "Speed (% of real time)"\
           …\
] -side right -fill y
```

Note that you still need the line continuation indicators (backslash) but if you do not like the asymmetrical placement of square brackets you may align them by column.

Configuring Widgets



- Changing configuration is possible at any time.
  - The widget's name is used as command
  - It is followed by the subcommand `config`,
  - which in turn is followed by the options

```
button .serverConnection\
       -text "Connect to Server"\
       -state disabled
…
.serverConnection configure\
    -state normal\
    -command connectViaTcpIp
```

Common options of all Tk widgets are centrally documented under the 'options' entry.

For widgets already known to the layout management, any changes to their configuration is promptly reflected in the GUI.

(Strictly speaking, the above is only true for interactive manipulation of widgets via Tk commands entered interactively. If widgets are created and configured – even when already packed – Tk may batch (and delay) update until the current handler is finished and it is ready to fetch the next event from the event queue, thereby eventually optimising updates.)



In the early days of Tk the idea of Geometry Managers was not in that widespread use as is today.

Basically it is centered around specifying a placing strategy instead of pixel coordinates for giving the widgets in a GUI a size and position. At first, this may seem an extra effort ans there is a large group of developers who may prefer a graphical tool to arrange the widgets of there application.

But strategies have clear advantages in at least two situations:

- The widgets actually displayed may change dynamically.
- The user should have the option to size the main window manually.
- The screen size is not known and may very substantially.

When PCs with screens of 800×600 or 1024×768 were dominant and application typically took over all scree estate the latter two were of little importance. Only recently GUI designers started to recognize that users may want to run applications on devices that radically differ in their properties, especially screen size.

This has lead from “*best viewed with a screen resolution of 1024×768*” (or similar), which was not unusual to be found on web-pages some ten years ago, to what is called “responsive layout” now.

Tk's layout management supports this easily:

- On the large scale you have the option to check the size of an application's main window and chose an optimized overall layout.
- For fine tuning you chose an appropriate layout management strategy.

Done!

(Your application always has a “nice” look & feel.)

Communicating with the Window Manager



- The window manager is responsible for:
  - Decoration of windows with a frame and a title
  - Standard buttons (for close, iconify etc.)
  - Moving and resizing windows including size constraints like a certain aspect ratio

```
wm title . "You cannot close me!"
wm protocol . WM_DELETE_WINDOW {}
```

As the window manager is a software component typically made available by a graphic user interface, i.e. outside Tk, many details of the `wm` command depend on the operating system.

Please refer to the Ousterhout Book (or any other decent Tk reference) to learn about more options. What is shown in this presentation are just a few typical or frequent use cases.

413
et seqq.

Binding to Events



- Events are at the heart of a GUI programming.
  - Standard behaviour is associated with most events
  - The command `bind` allows to adapt this to specific requirements

```
button .shy\
    -text "try click me"
bind <Enter> .shy {
    pack forget %w
}
```

This button cannot be clicked!

Know what? You're right!

Event binding is covered here but should not be overestimated:

- The default bindings of most are fully sufficient and
- Only very seldom they need not be changed or extended.

In fact, the author of this presentation has written a lot of Tcl applications with a GUI without making any use Tk's `bind` command – but sometimes it can add the icing to the cake.

Time for some Practical Exercises



- Review and run the program `tk_demo.tcl` as example for a very simplistic Tk application
- Review and run the program `101_table.tcl` for a slightly more challenging Tk application
- Note the program `keycodes.tcl` as a handy helper giving you the event names you may sometimes need for `bind`-ing.

Actually, until now not enough of Tk has been covered so that you could start to write a small “interesting” Tk application, so your assignment is not to write a silly “dummies”-application.

But you are encouraged, while reviewing the supplied examples, to make any changes that could help to test your understanding, especially:

- Widget configuration options.
- Layout manager options.

Discussion of Practical Exercises



- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Widgets and layout management have received no real coverage until now, because they are the topic of the next chapters. If any questions have come up

Nevertheless feel free to contribute what you have learned or observed by reviewing, running, and maybe modifying the examples for this chapters … and as always: any questions are welcome, though, this time, the answers might be right ahead after turning a few pages.

Part 8
Introduction to widgets

- What's ahead?
  - Simple Widgets
  - Complex widgets
  - Menus and Dialogues

Please consider this presentation as kind of golden thread or guided city tour, to give you a first, cursory view of the points of interest.

Especially if you come back to these pages later, be sure to have the Ousterhout book at hand (or any other decent Tk reference), as for doing any serious work you will surely need it.

Displaying Information in a `label`

- Typical uses:
  - Add a description to some other widgets
  - Display small amounts of varying information

```
proc x - { global a b r; set r [expr $a*$b] }
scale .a -orient horizontal\
    -from 1 -to 10 -variable a -command x
scale .b -orient vertical\
    -from 1 -to 10 -variable b -command x
label .r -font {Arial 30} -textvariable r
label .m -background darkgrey -text "x"
grid .m .a
grid .b .r
```



If you want to display varying information in a label, there are two basic ways:

- Only once connect the label with the `-textvariable` option to some global variable and change its content to display something different.
- Each time (re-) `configure` the label and give a new value for its `-text` option.

Hint: If you only want to change the text itself you may prefer the first way. If you also want to change any another label property along with the text it displays, like its foreground or background colour, or its font or font size, or whether the label is displayed sunken or raised etc., in the second way you can do all these changes with a single `configure` command.

Starting an Action with a `button`

- The uses:
  - Always start the same action or an action that depends on information supplied via other widgets
  - In the second case the button might stay disabled until that information is complete and verified

```
button .ok -label OK -state disabled
bind <… changes of other widgets …> {
    .ok configure -state\
        [lindex disabled normal [checkFormData]]
}
```


Using the `lindex` command to switch between two option values depending on a binary (true/false) value, as shown above, is a nice technique worth to keep in your bag of tricks.

Supplying Information via a `checkbutton`

- Typical uses:
  - Supply some “yes / no” information
  - Coupling an action on change is optional

```
checkbutton .showDebugOutput\
   -text "Show Debug Output"\
   -variable show_debug_output
```

Coupling a widget with a variable – where supported – is “two-way”: i.e. the variable may change depending on user interaction with the widget and the widget may change its appearance when variable is modified.

Because of its two-way connection a `checkbutton` widget is the ideal means to visualise and change binary options controlling the behaviour of some application.

Supplying Information via a `radiobutton`

- Typical uses:
  - Supply some mutual exclusive values
  - Buttons are associated by using the same variable

```
foreach rb txt val {
    .noDebug "No Debug Output" none
    .terseDebug "Terse Debug Output" terse
    .fullDebug "Extended Debug Output" full
} {
    radiobutton $rb -variable debug_output\
                    -text $txt -value $val
}
```


Because of its two-way connection a group of `radiobutton` widgets is a good means to visualise and change multi-valued options controlling the behaviour of some application.

Be sure to understand that e.g. for three different values you need three `radiobutton`-s, and note how all of them are created in the example above in a loop controlled by Tcl's `foreach` command.

Getting Information via an `entry`

- Typical use:
  - Request a single line of free text
  - An optional validation step may be linked on each character typed or when moving another widget

Be careful with validations that depend on other entries. E.g. an overly eager date input validation with three entries for day, month and year might make it make unnecessary hard to go from February 28th to March 31th ...

Because of its two-way connection an `entry` widget is a good means to visualise and change text options controlling the behaviour of some application.

If the set of acceptable values is restricted, there are several ways to validate user input before it gets effective. Though, validations with “corrections” must be done with some care to avoid endless loops. Be sure to have fully read and understood the relevant sections of the `entry` widget reference on validation and maybe write some small example programs to really get familiar with all the implications.

More Ways to Get Information

- Widgets to get constrained information are:
  - `scale`: a slider for horizontal or vertical placement, besides its range with various options, e.g. caption and tick-marks as well as resolution
  - `spinbox`: a dial control mainly for – but not limited to – numeric input, with an option whether to block or wrap-around at its limits
  - `listbox` and `combobox`: useful for selecting from several possibilities to chose from, in that similar to `spinbox` for value lists or `menubutton`

This page lists some more widgets you may use either to visualise or to control application state or both. There is no general rule which one prefer over the other – this is rather a matter of taste. Also user expectations need to be taken into account.

As a side note and general hint for GUI design: If possible, use the application you design yourself for some time for the typical work for which it is intended. Or, if you are not familiar with that kind of work, try to get some typical users to test a beta version and watch closely for his or her typical use patterns, especially for repetitive tasks which may be made more effective.

The more users later will work with the application (via that GUI, as we talk about GUI design), the more it pays to make its use more effective. Here is a concrete example:

Say you can make a typical task more effective by 12 seconds.

Doesn't look like a big deal, does it?

- Take a user base of only 100 (may be little or much, it depends) ...
- … assume each one can profit from the optimisation about thrice a day …
- … but he or she will use the application on average every second day only …
- … 45 working weeks in a year.

Now let's do the math:

0.2 (minutes) × 100 (users) × 3 (per day) ÷ 2 (every 2nd day) × 5 (days per) × 45 (weeks)

Gives 6,750 minutes or 112 hours!

If you spend that much time to find and implement this optimisation, it will amortize in the 1st year (if you and the assumed users are paid a comparable salary).

So, applications with a poor user interface are big suckers in our economy! Interesting, isn't it?

Interacting with `text` Data

- Typical use:
  - Display several lines of text
  - Optionally interact with that text in various ways

The `text` and `canvas` widgets are very rich in features – both will be revisited and covered in more detail later.

```
# there may be many lines, so add a scrollbar:
pack [scrollbar .sb] -side right -fill y
pack [text .t] -expand 1 -fill both
.t configure yscrollcommand ".sb set"
.sb configure -command  ".t yview"
```


At a first glance Tk's `text` widget may appear to be just a multi-line version of the `entry` widget, but actually it is far beyond that: It offers plentiful ways for text rendering and editing.

Though, its learning curve is very shallow, i.e.:

- With little knowledge you can do most basic things.
- With moderate knowledge you can do the more complex things.
- With expert knowledge you can do very sophisticated things.

If you can spare the time: get a good book on Tcl/Tk, one that covers the text widget in depth, and skim the examples. It's worth it!

433
et seqq.

Drawing to a canvas

- Typical use:
  - Draw and modify graphic shapes

If you need really sophisticated graphics capabilities in your Tcl/Tk application you might want look at `www.tcl3d.org`

```
pack [canvas .c]
.c create oval 30 30 120 120\
   -fill yellow -outline black
.c create oval 50 55 70 60\
    -fill blue
.c create oval 90 55 100 65\
    -fill blue
.c create arc 50 50 100 100\
    -style arc -start 200 -extent 140\
    -width 5 -outline red
```



The canvas widget too may be underestimated by beginners: it's a versatile drawing board with surprising capabilities.

On the other hand, depending on what you want or need to do, you may reach its limits sooner or later. If you want to deal with 2D-grahics on a higher abstraction level than simple lines, polygons, and arcs, or shapes build from these, it is to basic to be convenient.

The good news is that there are many Tk extensions, e.g. for data plotting, or, like Tcl3D, to manipulate three-dimensional objects and view the scenery from any perspective.

Introduction to Menus

- Typical uses:
  - Permanently displayed menu labels
  - Pulldown menus, expanded when a parent entry is selected
  - Popup menus, displayed on mouse click, maybe with content depending on context

Adding menus to a Tk application usually means to write a lot of boiler-plate code. Luckily it's easy in Tcl to roll your own small DSL (= *domain specific language*), which definitely helps to reduce bulky menu trees to their core and hence should be strongly considered.

Tk also supports the most prevalent menu styles, like:

- Pull-Down
- Pop-Up

As using the built-in menu support requires much (systematic and often similarly repeated) boiler-plate code, it pays to come up with schemes to generate menus in loops from a more abstract description.

(Until now the author had always run his own schemes at that point, simply looking for the repeated patterns and then re-packaging those into `foreach` loops controlled by table-like data structure. But there exist are also ready-made menu generators for Tcl/Tk.)

337
et seqq.

Standard Dialogues

- For some typical cases dialogues are available:
  - Confirm information or give final chance to execute or cancel an action (OK, Yes/No, Yes/No/Cancel, ...)
  - Allow selection of files, directories, colours ...

If only Tk wouldn't deliberately castrate my window system by giving me much less options than I usually have, or a completely different look&feel, e.g. for file selections!

Well, the difficult trade-off here is portability, so sometimes the least common denominator of all supported platforms dictates what can be made available ... but in many cases Tk does its best and will also use native controls where available, at least as an option.

The use of standard dialogues makes sense because – at least under windows – Tk uses those that the user expects and already knows to handle.

Unluckily there by far is not so much a standard style under Unix and Linux, e.g. when it comes to browse directories and select a file for input or output. Though Tk here gives a small set of choices, it might not contain what satisfy the user expectation. (This depends of course on what actually is expected and how flexible the user base is in case of smaller or larger deviations from what they are used to.)

Hint: As bringing up dialogues is not a too frequent operation and a time will be largely determined by bringing up the new window, depending on the application and its typical user base it may be worth while to consider running a small, external helper application with the expected look & feel, instead of knocking down unwilling end-users with something they do neither know nor like. This is especially true for the file selection dialogue, which is the by far the most frequent one a user has to cope with.

Toplevel Windows and Custom Dialogues

- New windows can also be created at top-level.
  - This does the `toplevel` widget
  - Also one of it may grab keyboard and mouse to force the user to interact solely with this top-level

Note that Tk has to step on “window manager territory” here and there *may* be non portable effects, e.g. if the top level is (created) withdrawn despite the fact it has a grab and the whole application becomes seemingly frozen.



To the author of this pages it appears as if a substantial set of application designers, if not the majority, have problems to understand one of the biggest advantages of a GUI:

- It is that it puts the user into full control over his or her next action.
- **And this should not be needlessly hampered by modal dialogues!**

Time for some Practical Exercises

- From the various demo programs (demo_*.tcl) select those dealing with the widget you are interrested in most, then run the the program.
- Optional: Change the program `widget_primer.tcl` so that it displays the result of an other mathematical operation but addition (say: multiplication, division …)
  - Try it out “fixed” first, i.e. just replace addition by something different in the source …
  - … then make it “selectable” by adding appropriate elements to its GUI

Note that the last step is not quite for the faint-hearted, as with respect to layout management not much has been covered until now. So you may set the goal to get the functionality right and defer the “attractive visual appearance” until later.

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Which one of the following did choose for the last part, i.e. to make the operation selectable?:

- A `radiobutton` group?
- A `listbox`?
- A `menu`?
- A `spinbutton` with a given set of values?
- An `entry` with custom validation?
- Something else?

Which one of the above, do you think, hypothetical users of the program might like the most?

Part 9
Layout management



- What's ahead?
  - The packer and the gridder
  - Frames and scrollbars
  - Split and tabbed panes

As often with Tcl/Tk, the “learning by doing approach” or better “learning by trying variations” works very well to understand layout managers. What you need to prepare are a number of widgets to play with. Besides different default lengths using different background colours is advantageous, so here is a trick to create a bunch of labels with a simple loop:

```
foreach v {
    aquamarine burlywood cyan darkgrey
    firebrick green hotpink ivory
} {
    label .[string index $v 0] -text $v -background
$v
}
```

Note that only the first character of the colour name is used as widget name, because you may need to it frequently. Also, the colour names have been chosen to differ a sufficiently in their visual appearance. (But sorry, no, Tk has no colour name with an “e” as its first character.)

Also be aware of the *Shrink-Wrap-Behaviour* of Tk's main window!

When starting `wish` a Tk main window appears in a size chosen by the window manager – a component of the operating system, not Tcl/Tk. Its size will change once you start `pack`-ing or `grid`-ing widgets (according to what is following) to accommodate the space requirements, until you resize it manually for the first time and will keep its size from now on, also if this means that one or more widgets get less space as they would require. E.g. a label now might not any longer display its full text.



With the a set of label widgets as proposed at the start of this chapter you may interactively try things like the following and watch their effect:

- `pack .a -side left`
- `pack .b -side left -fill y`
- `pack .c -side top -fill y`
- `pack .d -side bottom -fill both -expand 1`
- … (etc.) …

You can also remove packed widgets, e.g:

- pack forget .a .c

As layout management is also about defining a strategy that makes a GUI looks good or at least acceptable in case of different window sizes, resize the TK window for the layouts you try.

This is especially important to understand the effect of the `pack` option `-expand 1`.

To re-establish shrink-wrap-behaviour the following command can be used:

- `wm sizefrom . ""`

(Note the dot preceding the empty string argument.)



With the a set of label widgets as proposed at the start of this chapter you may interactively try things like the following and watch their effect:

- `grid .a .b  - .c`
- `grid .d  ^  ^ .f -sticky we`
- `grid .g .h .i  - -sticky news`

You may also want to watch the effect of row and column weights when resizing the main window:

- `grid columnconfigure . 0 -weight 1`
- `grid rowconfigure . 0 -weight 5`
- `grid rowconfigure . 2 -weight 2`
-

Be sure to include the dot (referring to Tk's main window) left of the row or column index.

(When trying out things interactively you may want to abbreviate long sub-commands. E.g. instead of `columnconfigure` you may also use `columnconf` or even `columnc`.)

As before you may remove widgets with the `forget` (sub-) command.

Also the following may come in handy

- `eval grid forget [winfo children .]`

to make “tabula rasa”, i.e. remove all currently displayed widgets.



This kind of nesting is the key to complex screen layouts: Whenever it seems too difficult to get what you want with the sole use of pack or grid, think of inserting a frame and combine both, though there is a pitfall:

It must be made absolutely sure that each widget is managed by only <u>one</u> layout manager!

(Otherwise layout managers may start to fight with each other when sizing the widgets they manage, causing all kinds of strange effects, including hanging or crashing Tk completely.)



Not only as a question of style but also as a matter of user interface economy, tabbed panes should be considered when there is

- too much information as fits into a single window,
- which can be combined into related groups,
- of which one should always be completely visible,
- while arbitrary switching between groups is desirable.

If only some but not all of the above is primal, usual alternatives are scrollable windows and series of modal dialogues. (Both are also possible with Tk.)



Not only as a question of style but also as a matter of user interface economy, paned windows should be considered when there are

- two or more “working areas” are required to perform a certain tasks,
- with typically one is the main working area and should receive most space,
- but which one can not be decided in advance, e.g. because it varies with progress.

A typical alternative were distinct top-level windows, which are also possible with Tk.

Note: Until Tk 8.4.20 there is no “out of the box” support to freely change to the overall arrangement of paned windows (as e.g. Eclipse allows). Only recently – starting with Tk 8.5 – basic support to dock and undock (sub-) windows has been added.

Absolute Placement and Sizing of Widgets



- There is also a very basic geometry manager.
  - It is commonly called “the placer”
  - Communication with it is via the `place` command
  - Widget layout is based on pixel coordinates
  - Use cases of the placer are as foundation for more sophisticated geometry management …
  - … or in special applications like embedded devices



As a personal note: many of the GUIs the author of this presentation has built based on Tk were at least partially "data driven", meaning that the exact content (I.e. the widgets to be created and placed) was not fully before runtime.

As an example consider a log-file filter that allows to display and hide messages depending on a tag which might indicate there severity or a module to which they refer and the set of severities or modules is not known at design time, nevertheless you want check-boxes depending to display or hide the categories you want or don't want to be displayed.

In that case you could still build a GUI programmatically, while you see which severities or module names actually are present in the log-file.

At the bottom line it has been mostly found easier and more productive to "generate" the GUI from a more or less abstract description, driving Tcl's `foreach` through data structures. Particularly Tcl's `dict`-s have been found helpful here, as they lend themselves easily when it comes to create "domain specific languages" (DSLs) with a certain degree of recursive nesting.

Time for some Practical Exercises



- Effective use of layout managers is mostly a matter of practical experience. So take your time to test and try Tk's `pack` and `grid` command interactively first until you feel confident enough procede.
- Continue with the program `widget_primer.tcl` in the state of development it had after the last chapter.
  - Design and implement a “nice” layout using the Tk's `pack` command for layout management.
  - Design and implement a “nice” layout using the Tk's `grid` command for layout management.

You also may also switch the order, i.e. first create “nice” Tk GUI with `grid`, then with `pack`.

For those who are rather interested reviewing examples, there are also two demo programs provided:

- `packer_demo.tcl` to print a table of trigonometric functions with a GUI using the pack command for layout management.
- `gridder_demo.tcl` which is similar but uses a mix of pack and grid commands (with `frame` widgets as intermediary).

(Though, judging from the authors own experience, while demo programs for layout management can give some initial ideas, it pays much, **much** more to gain “hands on” experience by trying the layout managers interactively as suggested throughout this chapter.)

Discussion of Practical Exercises



- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

What would make especially sense here is to request (and discuss) explanations if some layout manager behaved in ways which you cannot explain yourself. Because, as long as this happens, you probably have not yet developed enough understanding to avoid surprises, when you have to walk alone.

Part 10
Text and Graphic



- What's ahead?
  - The tree widget
  - More features of the text widget
  - More features of the canvas widget

The topics of this chapter can be covered in two ways. (Which one to take may also depend on your general interest in GUI and the time left.)

- As an overview, more or less simply presenting some few more options.
- Guided by ideas for concrete applications you'd like to build with Tk after this course.

In the latter case advanced widgets could now be covered more focused to your needs, and during the practical exercise you could also start to build a rough prototype.

Hierarchically Structured Information



- The `ttk::treeview` widget is designed to:
  - Display and work on hierarchical information
  - Of course including the option to expand and collapse nodes to hide or show details
  - This can go to any depth and may be different for any node

"Learning by Doing" is a good approach here, so take the time, open some reference documentation – online or in print – run a `wish`, create a `ttk::treeview` and start playing with it interactively or with small code snippets you enter in a text editor …

Advanced Text Processing



- The `text` widget is extremely rich on features:
  - Without exaggeration it may be placed in the category of a simple or even moderately advanced text editors
  - Among other things it knows about *marks* and can *tag* ranges of text.
  - Though its most striking feature may be the ability to hold embedded widgets of nearly any kind.



Besides marks you may set yourself, there are more “automatically” set marks besides the above:

- `1.0` – insert into a text widget before the very first character
- `end` – insert (or rather append) in a text widget after the very last character

What is sometimes confused is the difference between

- `insert` – a distinct location in the text visually marked with the *Insertion Cursor* when the `text` widget has the input focus and
- `current` – the location in the text closest to the *Mouse Cursor*

The confusion may partially be blamed of the fact that the default bindings causes both locations to become identical, nevertheless they are generally distinct concepts.

Tagging Text



- Assigning a tag to text spans is very powerful:
  - Reference to that range is via a symbolic name, that keeps valid in case of insertions or deletions
  - Any number of text spans can share the same tag
  - Vice versa, differently tagged ranges may overlap

Tagging text can also be seen as semantic mark-up:
actual presentation including
- font, size, and style (regular, bold, italic, underlined …),
- foreground and background colour,
- the option to elide the tagged span,

and more is assigned to the tag, not the characters in the text themselves, and therefore is easy to modify centrally.

From his personal experience the author of this presentation can contribute that semantically tagging text is a very, **very** powerful feature and sometimes provided an easy solution to what at first glance seems a complicated pogramming task.

Binding Events to Tagged Text



- Text tags may also assume bindings:
  - This includes but is not limited to mouse clicks
  - That way a text widget's content could function similar to hypertext
  - Of course, any action may follow a click – including but not limited to displaying some other text page

Default bindings exist so that a text span selected by marking with the mouse or using the cursor keys with SHIFT held down is automatically tagged with `sel` and can easily be copy-pasted to different location, to some other text window or even to another application.

This feature provides the basics to embed user interfaces more or less “in the text flow”, like found in web pages. Besides allowing to trigger actions by clicking on words, you might e.g. also embed `ckeckbutton`-s or `menubutton`-s etc.

- Pro: the general look and feel is “text-flow” oriented, which might be a benefit e.g. help text that is typically percepted sequentially.
- Contra: There is much lesser order imposed as e.g. in a tabular layout, where control elements are arranged more or less columnar.

Embedding other Widgets into Text



- The content of `text` widgets is not limited to characters, it can be any other widget too!

```
pack [text .t]
checkbutton .t.ok -text OK -variable ok\
                  -command {colorize $ok}
.t insert end "A checkbutton here?" bgc " "
.t window create end -window .t.ok
proc colorize {x} {
        set red_green [lindex {red green} $x]
        .t tag config bgc -background $red_green
}
colorize 0
```

Advanced Canvas Features



- The canvas widget is very powerful too:
  - Individual items of basic shapes can be tagged
  - Items sharing a tag can be handled as a group
  - This includes moving, resizing, and binding events
  - Also supported is the detection of overlapping shapes and the calculation of bounding boxes.

Even on the danger of repeating myself: For the `canvas` widget too, “Learning by Doing” is a recommendable approach to get comfortable with its basic and advanced features.

Basically Tk's `canvas` can be characterised as:

- A container for basic graphic primitives like
  - various kinds of straight lines (including polygons),
  - various kind of bended lines (circular arcs, bezier curves …),
  - planar shapes (rectangles, closed polygon, ovals/circles)
- with certain attributes for
  - line thickness and colour, fill colour for planar shapes, etc.
  - line endings and connections (flat, round, bevelled, etc.)
- and the option to
  - visualize the primitives fully or partially
  - if they lie inside a given view-port

Canvas Scrolling



- A `canvas` actually is a virtual drawing area.
  - It's size is at least 32000 × 32000 pixel
  - The part visible on the screen may be smaller than the space required by a drawing in the canvas
  - Tcl's standard techniques for scrolling the canvas content apply with respect to moving the viewport
  - Commands for translating between canvas and screen coordinates are available

Note: As the internal storage is “vector-based” any scaling will be lossless.

Canvas Printing



- A canvas can also be prepared for printing.
  - The standard way is to convert its content to EPS
  - This is done with the sub-command `postscript`
  - The result can be stored in a string or in a file

EPS stands for Embedded Postscript. It was a portable standard when Tcl/Tk was designed, but is a bit antiquated today. As a more modern alternative you may want to have a look on the (free) extension `pdf4tcl` (see: `pdf4tcl.berlios.de`).



After exporting to a PostScript™ representation also external tools may be available to further convert a canvas drawing to most any prevalent graphic format.

Time for some Practical Exercises



- If you have no other ideas, try yourself on a simple Log-File Viewer based on Tk's `text` widget:
  - Assume a text file format which can be parsed into
    - Header (e.g. time stamp, severity, originating module)
    - Numerical Error Code
    - Textual Error Description
  - The viewer should display the lines colour code according to their severitiy and allow easy selection (display/hide) based on a time window and the originating module.

Besides GUI programming with Tk you can try some Tcl:

- Reading from a file
- Parsing lines of text

For the latter you may use regular expressions to match the following fields (separated by space):

<*time-stamp*> <*severity*> <*module*> <*error-code*> <*error-text*>

With:

- <*time-stamp*> YYYY-MM-DD hh:mm:ss
- <*severity*> one of ERROR WARNING INFO
- <*module*> any alphanumeric string
- <*error code*> any integral number
- <*error text*> anything following (up to the end of line)

Build the program step by step, i.e. start with a prototype that reads a file with fixed name name and displays the content in a text widget, then add colour coding, a set of `checkbox`-es to filter for modules (depending on what appeared in the actual input).

If all of this works you may proceed by adding a file chooser dialogue for selecting a log-file, `spinbox`-es to set the limits of a time window for further filtering.

Of course, you can add more icing on the cake, e.g. user configurable colour coding – yes, there is also a colour chooser dialogue in Tk.

Or have the viewer receive log-messages via a TCP/IP-socket.

But try not to get carried away by a single pet feature you'd like to have – get an understanding for the basic techniques first, the fine grained details may wait until later, after this course.

Discussion of Practical Exercises



- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Filtering for module names and severity is basically similar. How much effort would it be to add the latter to your solution.

Finally, as Tk's `canvas` widget wasn't covered by that exercise at all, here is an idea for a home assignment: Add some graphical statistic evaluation to your log-file viewer:

- Display the relative frequency of *errors*, *warnings*, and *infos* as pie-chart
- or as (coloured) blocks or spikes along a time line.

Part 11
Miscellaneous Features

- What's ahead?
  - Introspection and tracing
  - Some hints on structuring applications
  - Object oriented Tcl
  - Extending Tcl with C/C++-modules
  - Tcl integration in XILINX tools

This chapter covers a mix of features that may be useful at times.

The selection is not quite arbitrary but draws much on the experience of the author of this document and is therefore closely connected to the field of his works. In different fields features not mentioned here may be valuable too.

Do not take this chapter as exhaustive coverage of “everything else” Tcl/Tk has to offer.

Introspection

- The `info` command gives access to:
  - Names of variables
  - Names of subroutines, including parametrization
  - Implementation of subroutines (only if code in Tcl)
  - … (and much, much more) ...

Besides occasional interactive use the above features are valuable for building debugging support, i.e. they may be most valuable for authors of debugging tools, or to understand the inner workings of a non-trivial, scarcely documented Tcl (/Tk) application written by someone else.



As especially write-traces can not only check but also change what is assigned to a variable, it may also be used to impose constraints on variable values.

By means of execution traces it is also possible to systematically insert code before and after existing subroutines (that stay unaltered so far, much in the same way as with aspect-oriented programming).

Separating Subroutine Defintions

- The `source` command allows to:
  - Read parts of an application from separate files
  - Typically these files will contain just subroutine definitions
  - Double inclusion then is harmless (maybe except some minor performance loss)

If you work at the `tclsh` prompt you may also easily re-read a Tcl source file that way after some change in an editor, as a new function definition silently replace an existing one.



Tcl's `source` command is typically the first step going from “single source file” applications into the direction of (partially) reusable code.

Autoloading

- The auto-loading facility is useful to:
  - Automatically read the file containing the definition when a subroutine is called for the first time
  - Start-up time may be slightly reduced as loading the source files is delayed unti first use
  - On the downside, in case of a miss-configuration (incomplete installation etc.) data dependent errors may occur at runtime.

Autoloading is often the next step (after explicit use of the `source` command) if a Tcl (/Tk) application grows with time, there are more and more (reusable) parts and dependencies start to get hard to manage manually.



If it can be foreseen that a Tcl (/Tk) application will be used and extended over a longer period of time, and probably grow large, autoloading might be skipped in favour of using the package facility right after the prototype outgrows the “one source file” state.

The main advantage of packages is the option of versioning in case of incompatible changes: several versions of a package may coexist and be loaded distinctively.

If the package facility is used for versioning, be sure to understand the following:

- Loading different versions of a package into separate applications (stored in source files executed by different instances of `tclsh` or `wish`) will never cause problems.
- Otherwise care must be taken to avoid accidental name clashes, e.g. by precautionary use of namespaces.

Extending Tcl with C/C++ Modules

- Tcl stands for Tool Command Language
  - This was it was John Ousterhout originally had in mind: a language core to be extended by tools
  - Nearly 200 of 700 pages in his book cover this topic: Extending Tcl via its C API
  - To summarize: It is easy to implement new Tcl commands in C or C++ as long as you accept to write a certain amount of boiler place code

As expressed in its very name, the original idea of John Ousterhout when he created Tcl a quarter century ago was to provide authors of other tools with a *Tool Command Language*.

Extensibility therefore is not a by-product but once was at the core of Tcl's design goals.

Tcl and SWIG (see: www.swig.org)

- SWIG is a tool developed at NASA:
  - It is a *Simplified Wrapper Interface Generator* to ...
  - produce boiler code based on a description ...
  - in a syntax much resembling a C/C++ header ...
  - for glueing C/C++ routines to a language like Tcl

The words that should have triggered your attention are "***like Tcl***" because once you have the C/C++ routines and the interface file it's just a different SWIG command line option to glue the same routines with: *Allegro Common Lisp, Android, C#, Chicken, D, Go, Guile, Java, Common Lisp, Lua, Modula3, MzScheme/Racket, Occaml, Perl5, PHP, Pike, Python, R, Ruby …* and surely more to come.

The "quick-start tour" for Tcl part of the SWIG documentation does not take more than an hour and will give a good idea about the effort – or: how little effort – it actually takes to integrate Tcl with C/C++.

Go and take this tour (if this is a topic of interest).

Object Oriented Tcl

- There are several object oriented extensions:
  - *Itcl* aka *[incr tcl]*:
    makes most C++ concepts available in Tcl's syntax but requires an extended `tclsh` / `wish` (`tix`)
  - *xotcl*:
    supports selected object oriented concepts in Tcl
  - *snit*:
    an alternative to *Itcl* written in pure Tcl

Software developers with a solid background in both Tcl and C++ will probably feel comfortable with [incr Tcl] at a very shallow learning curve.

The open nature of Tcl and the ease with which new commands can be added has produced several extensions to “allow or improve Object Oriented Programming”.

The most ambitious – which also did also forestall the namespace feature later integrated into main-stream Tcl – was Incremented Tcl, or [incr tcl]. It looks and feels like “C++ with Tcl Syntax” (and less strict type checking as it is in the nature of scripting languages).

While *[incr tcl]* uses a modified interpreter, the “just another package to load” approach of *snit* is at the other end of the spectrum.

(The author of this presentation has never worked with *xotlc* and hence cannot comment on it.)



Beyond the above nothing needs to be understood to get the broad picture about the integration of Tcl with specific tools.

(Where it can get “interesting” in the meaning that more or less challenging problems arise and need to be solved by the integrator is when it comes to event-driven programming and Tcl's event loop must be merged with another such loop.)

Tcl in Design Tools (General Picture)

- Tcl based design tools are not more, not less as common Tcl **plus tool specific commands**!
- What does it take to drive a taxicab in Berlin?
  - You must know how to drive a car
  - You must be familiar with Berlin's road map
- What does it take to use Tcl-based tools?
  - You must know how to use Tcl as language
  - You must be familiar with the tool specific commands

Right from the start Ousterhout's target group were design tools chip design, i.e. integrated semiconductor devices.

So, the actual “by-product” is not that Tcl is used today in tools like Vivado but that it turned out to be useful for so many other kinds software; and though nowadays Tcl/Tk is not “cutting edge” technology, it has proven to a robust base across-the-board.

Tcl in XILINX Design Tools

- Tcl is part of several XILINX tools:
  - *Vivado*, *ISE*, *Impact, ...*
  - Tcl's syntax is unchanged in general
- Except in Vivado where square brackets do not cause command substitution **if and only if**
  - enclosed is an asterisk, i.e. `[*]`, or
  - a number or range, e.g. `[12]` or `[0:15]`

Interestingly the above has a not too complicated solution: Vivado simply searches command lines for the above pattern and handles these before it comes to recognizing square brackets as command substitution in Tcl's regular syntax parsing.

Time for some Practical Exercises

- If you want to try something from this chapter and have no idea what it could be, feel free to use the suggestion below.
- Also it is perfectly OK if you continue with some not quite finished exercises from earlier chapters or a personal “mini project”, should you have started one.
- Finally – if the majority rather want's this – we can together take a tour through more practical examples of Tcl/Tk programming.

Implement a variable to store positive integers (`1`…) only.

- Start with a prototype that adds a write trace to some variable which checks the assigne value and throws an error if it is not a positive integer.
- Design a convenient scheme to create such variable, maybe some subroutine (suggested name `posint`) to which the variable name is given as argument.
  - That subroutine should create the variable and if there is an optional second argument it should be used as initial value:
    ```
    posint x 222   ;# create in caller's scope
                    # and initialize with 222
    ```
  - Otherwise the initial value should be 1:
    ```
    posint y      ;# create y in caller's scope
                   # and initialized to 1
    ```
  - Invalid initial values should be considered as error
    ```
    posint z abc  ;# do NOT create z
                   # but throw an error instead
    ```

In a similar style as above, implement a variable that can only store the seven days of a week as three letter abbreviations, i.e: `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, `Sun`

- ```
  weekday w1 Fri ;# create w1, initialise with
  Fri(day)
  weekday w2     ;# create w2, initialize with
  Mon(day)
  weekday w3 Mit ;# error
  ```
- Find some way to get into control when the `incr` command is called with such a variable and advance the contained day, with wrap-around at the end of the week
  ```

Discussion of Practical Exercises

- What have you learned so far?
  - Tell all of us!
  - Any clarifications necessary?

Would you still support the notion that Tcl's variables are untyped … :-)

Part 12
Open Box – You Choose!

- Maybe from:
  - More on practising event driven design
  - More on Tk concepts and widgets
  - Tcl's virtual file system
  - Tclkits and Starpacks

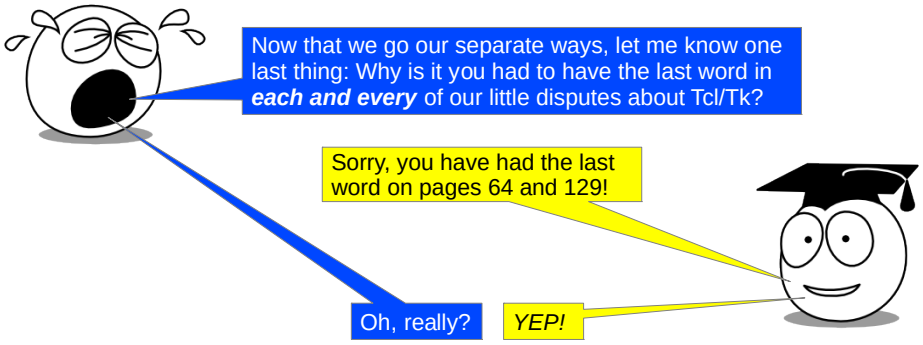
(No More Time for) Practical Exercises

- Probably at this point there is no more time for practical exercises – at least nothing that can be discussed after it has been completed.
- But if there is still some time and the majority decides we can take a tour through more practical examples of Tcl/Tk programming.

Here is a small overview of what has been supplied as auxiliary examples:

- (TBD)

Epilogue



Have fun with Tcl/Tk.