

A PRACTICAL TOOLS GUIDE · OFFLINE EDITION

# Tcl for EDA: From First Command to Flow Script

---

Commands, Lists, Regular Expressions, Dictionaries, Files and  
Flow Scripts, with Innovus Examples

Twelve lessons that take you from the first line of Tcl to a complete script that reads a report and decides what to do next. Each idea comes first, then the command that expresses it.

pdVerse Tools · Tcl for EDA

Training material. The Tcl examples were run in tclsh 8.6. The Innovus examples were not run on a licensed tool while this edition was prepared, so check every option in your own release. The lessons are free to read online at [pdverse.net](http://pdverse.net).

*explore more at [pdverse.net](http://pdverse.net)*

---

# Table of Contents

---

|                                              |     |
|----------------------------------------------|-----|
| • How to read this guide                     | 3   |
| • 1 Why every EDA tool speaks Tcl            | 4   |
| • 2 Commands, words and substitution         | 10  |
| • 3 Variables and expressions                | 17  |
| • 4 Lists                                    | 23  |
| • 5 Strings and regular expressions          | 31  |
| • 6 Control flow and error handling          | 39  |
| • 7 Procedures, scope and namespaces         | 48  |
| • 8 Arrays and dictionaries                  | 57  |
| • 9 Files, directories and external programs | 65  |
| • 10 Working with design objects             | 73  |
| • 11 Parameterised flow scripts              | 80  |
| • 12 Debugging, speed and a complete script  | 87  |
| • One-page summary                           | 96  |
| • Appendix A: Command cheat sheet            | 97  |
| • Appendix B: Exercise kit                   | 102 |



# How to read this guide

## NOTE

Each lesson names the idea first, then shows the command that expresses it. Lessons build on each other, so read them in order. The Tcl is standard Tcl 8.6, so it carries to any tool that embeds Tcl. The tool examples use Cadence Innovus.

Let us agree on how the boxes look, because the same few kinds are used throughout.

| Box                            | Looks like                                                | Meaning                                                                                  |
|--------------------------------|-----------------------------------------------------------|------------------------------------------------------------------------------------------|
| Black box, label "Tcl"         | White text on black.                                      | Plain Tcl. Type the lines at a tclsh prompt, or save them in a script.                   |
| Black box, label "Innovus Tcl" | White text on black, with Legacy UI or Common UI in gold. | Lines for the Innovus prompt. Check each option with <code>-help</code> in your release. |
| White box, label "Output"      | Dark text on white.                                       | What the lines above print. Where the result changes between runs, no number is shown.   |
| White box, label "File"        | Dark text on white.                                       | The content of a file in the exercise kit.                                               |

Lines that start with `#` are comments. They explain the command, and Tcl ignores them. Legacy UI means the camelCase commands such as `timeDesign` and `saveDesign`. Common UI means the `set_db` and `get_db` style. Do not mix the two in one script.

The plain Tcl examples were run in tclsh 8.6, and the outputs shown are what that run printed. The Innovus examples were not run on a licensed tool while this edition was prepared. They come from the Innovus command reference and the synthesis lab kit, so treat them as patterns to check, not as logs.

Lessons 1 to 3 cover how Tcl reads a line and how it handles values. Lessons 4 to 8 cover lists, strings, control flow, procedures and dictionaries. Lessons 9 to 12 cover files, design objects, flow scripts and debugging. Appendix A is a command cheat sheet, and Appendix B is the exercise kit.



## LESSON 1

# Why every EDA tool speaks Tcl

Open the command line of Genus, Innovus, Design Compiler, ICC2 or PrimeTime and you land in the same place: a Tcl prompt. The tool's own commands, such as `place_opt_design` or `report_timing`, are added to a language that already exists. This lesson explains why that choice was made, what you get from it, and the few ideas you must hold before writing a single script.

## 1.1 What Tcl is

Tcl (the Tool Command Language) is a small scripting language built to be embedded in other programs. A tool author links the Tcl interpreter into the application, registers the application's functions as Tcl commands, and gets a full scripting language without designing one. Users get loops, variables, procedures and file handling for free. The tool author gets one stable language that every customer already knows.

Let us consider what that means for you. When you type `setOptMode -opt_add_insts true` in Innovus, the Tcl interpreter reads the line, finds a command called `setOptMode`, and hands it the words that follow. The command belongs to the tool. The reading, the variables around it and the `foreach` loop you may put it in belong to Tcl.

## 1.2 Two places to run Tcl

You can run Tcl in two different places, and it helps to keep them apart.

1. **`tclsh`** is a plain interpreter that ships with Tcl. It knows the language and nothing about any design. This is where you learn and test your scripting logic.
2. **The tool prompt** is the same interpreter with extra commands. Everything that works in `tclsh` works here too, and in addition you can call the tool's commands.

Most of this course uses `tclsh`, because a script that parses a report or loops over a list of names does not need a tool. When a small tool example helps, it is shown in a separate block



and labelled. Check which Tcl version a tool uses by running `puts [info patchlevel]` at its prompt. The examples in this course were run on Tcl 8.6.12.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts "hello from [info patchlevel]"
```

This prints:

**OUTPUT** *(what the commands above print)*

```
hello from 8.6.12
```

## 1.3 Everything is a command

Tcl has almost no syntax. There are no keywords in the usual sense. `set`, `if`, `foreach` and `proc` are all ordinary commands that take arguments. A script is a sequence of commands, one per line (or separated by semicolons), and each command is a list of words. The first word names the command, and the rest are its arguments.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set period 2.0
puts "period is $period ns"
puts "half period is [expr {$period / 2}] ns"
```

This prints:

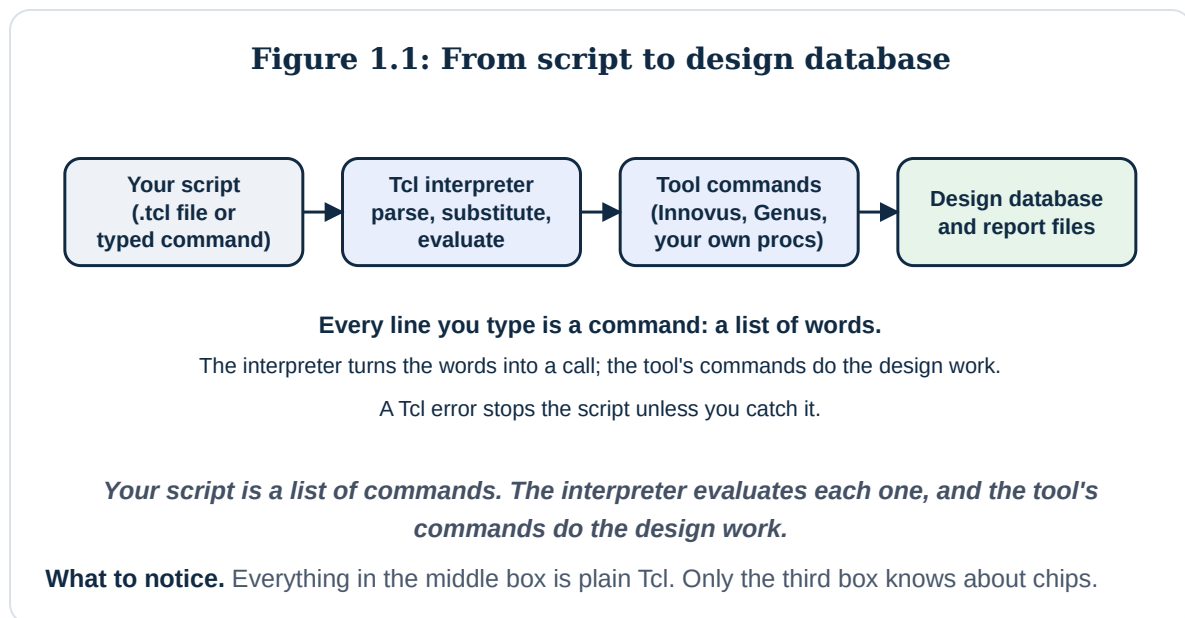
**OUTPUT** *(what the commands above print)*

```
period is 2.0 ns
half period is 1.0 ns
```

Notice three commands: `set`, `puts` and `expr`. The square brackets run a command and use its result as a word. The dollar sign reads a variable. Lesson 2 shows exactly how these work. For now, hold on to the rule that every line is a command and every command is a list of words.



Figure 1.1 shows how a script, the interpreter and the tool fit together.



## 1.4 Everything is a string

Tcl treats values as strings. A number, a list and a file name are all text, and the interpreter converts between meanings when a command needs it. A variable holding `10` and a variable holding the text `"10"` are the same thing.

**TCL EXAMPLE** (type at the `tclsh` prompt, or put in a script)

```
set a 10
set b "10"
puts [expr {$a == $b}]
puts [string length 12345]
```

This prints:

**OUTPUT** (what the commands above print)

```
1
5
```

The consequence for EDA is practical. Names of cells, nets and pins, numbers from reports and whole lines from a log all arrive as strings, and you can search and slice them with the same



small set of commands. The price is that Tcl will not warn you when you treat a list as a string or the reverse. Lessons 4 and 12 return to this.

## 1.5 A command is a list

Because a command is a list of words, you can build one as data and run it later. This is how flow scripts assemble tool commands from settings.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set words [list set clk_period 2.0]
puts [llength $words]
eval $words
puts $clk_period
```

This prints:

**OUTPUT** *(what the commands above print)*

```
3
2.0
```

The list holds three words, and `eval` runs them as a command. You will use this idea carefully in Lesson 11. It is powerful, and it is also the source of the most common quoting bugs.

## 1.6 Scripts, source and batch runs

A script is a text file of commands. You run it with `source`, which reads the file and evaluates each command in turn.

**Legacy UI.** To be run at the Innovus prompt. Check the options against your release with `source -help`.

**INNOVUS TCL LEGACY UI** *(to be run at the Innovus prompt)*

```
source -echo run_place.tcl
```

The `-echo` option prints each command as it runs, which makes a log easy to read. Without options, the global `enc_source_verbose` decides how much is printed.



For a run with no window, the Common UI synthesis lab flow starts the tool with the script as an argument. To be run from a shell, with the options checked in your release:

```
SHELL  COMMON UI BATCH LINE  (run from a shell; check the options in your release)

innovus -synthesis -stylus -no_gui -files run.tcl -log run1 -overwrite
```

The Legacy and Common UI command sets are different, but both are Tcl. A `foreach` loop is written the same way in each. In this course every tool example is labelled so you always know which command set a line belongs to.

## 1.7 What Tcl is not

Tcl is not a fast numerical language and not a type-checked one. A loop over a million cells in Tcl is slow compared with a command the tool runs in its own code. That is the reason tool commands such as `get_cells -filter` exist: let the tool select, and let Tcl decide what to do with the result. Lesson 10 returns to this balance.

## 1.8 What goes wrong without Tcl knowledge

A designer who copies tool commands from a colleague's script can run a flow. The trouble starts when a command needs to change. A hard-coded cell name needs to become a loop over many cells. A report must be read to decide whether to continue. A step fails in the middle of a night run and nothing recorded why. Each of those needs a few Tcl ideas: variables, lists, string and regular expression commands, `catch`, procedures and files. The rest of this course covers them in that order.

### 🔗 WHY AN INTERVIEWER ASKS ABOUT THIS

Physical design and CAD interviews often begin with Tcl because it shows how a candidate really works. Likely questions: "what is the difference between `tcsh` and the tool shell", "why is everything a string in Tcl", and "how would you run the same command over a list of cells". A good answer names the interpreter, explains that the tool adds commands to it, and describes a `foreach` loop over a list.



---

**✓ KEY TAKEAWAYS**

- EDA tools embed a Tcl interpreter and add their own commands, so Tcl skill carries across Innovus, Genus and the rest.
- A script is a list of commands, and every command is a list of words.
- Values are strings, and you convert meanings by choosing the right command.
- Tool commands do the heavy work on the design, and Tcl decides what to run and what to do with the result.



## LESSON 2

# Commands, words and substitution

Almost every Tcl surprise comes from one question: when does the interpreter replace `$name` or `[command]` with a value, and when does it leave the text alone? This lesson gives the whole rule. It is short, and once it is clear, the quoting problems that trouble EDA scripts become easy to predict.

## 2.1 Words

Tcl reads one command at a time. A command ends at a newline or a semicolon. It is split into **words** separated by spaces. The first word is the command name, and the others are its arguments.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set name acc8
puts "design: $name"
puts {design: $name}
puts "length: [string length $name]"
puts "a\tb"
puts "cost is \$5"
```

This prints:

**OUTPUT** *(what the commands above print)*

```
design: acc8
design: $name
length: 4
a    b
cost is $5
```

The second line printed the text `$name` because it sat inside braces. The fourth line ran `string length` and printed its result. The last two show the backslash: `\t` became a tab, and `\$` kept the dollar sign as an ordinary character.



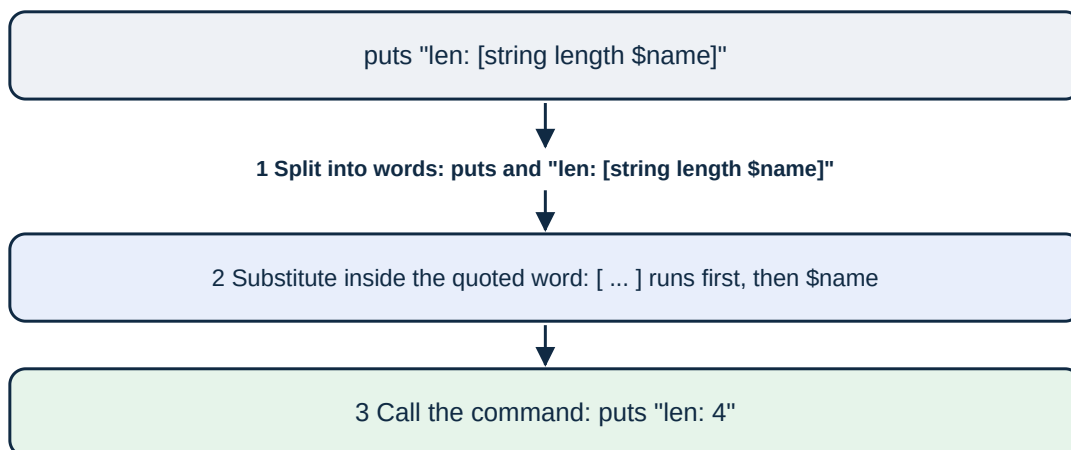
## 2.2 The three substitutions

After splitting a command into words, Tcl performs three kinds of substitution on each word, and only these three.

1. **Variable substitution.** `$name` is replaced by the value of the variable. Use `${name}` when characters that could belong to a name follow it.
2. **Command substitution.** `[command args]` is replaced by the result of running the command.
3. **Backslash substitution.** A backslash followed by a character is replaced by a special character (`\t` tab, `\n` newline) or by the character itself (`\$`, `\[`, `\\`).

Figure 2.1 shows the order for one line.

**Figure 2.1: How Tcl reads one line**



*Substitution happens once, left to right. Braces switch it off.*

***Substitution runs once, left to right, inside the quoted word.***

**What to notice.** Step 2 happens before the command is called, so `puts` only ever sees the final text.



Substitution happens once. The result of a substitution is never looked at again for more substitution. That is the reason this works as expected:

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set x 3
set y {$x}
puts $y
puts [subst $y]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
$x
3
```

The variable `y` holds the four characters `$x`. Printing it gives those characters. Only the `subst` command, which asks for substitution on purpose, turns them into `3`.

## 2.3 Grouping with braces and quotes

A word that contains spaces needs grouping. Tcl has two ways.

- **Double quotes** group the word and still allow all three substitutions inside.
- **Braces** group the word and allow none. What is between the braces is passed on exactly as written. Braces nest.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set a "one two"
puts [length $a]
puts [length {$a}]
puts [length "$a three"]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
2
1
3
```

The first call sees two words. The second sees the single word `$a`, with the dollar sign, because the braces prevented substitution. The third substituted `$a` inside the quotes and then counted three words.

A good habit follows from this. Use braces for code and patterns that should be passed on untouched, such as the body of a loop or a regular expression. Use double quotes when you want variables and commands filled in.

## 2.4 A worked bug: the name that swallowed the suffix

Let us consider a script that builds a gated clock name.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set clk_name clk
catch {puts "$clk_name_gated"} err
puts $err
puts "${clk_name}_gated"
```

This prints:

**OUTPUT** *(what the commands above print)*

```
can't read "clk_name_gated": no such variable
clk_gated
```

Tcl read the longest possible variable name, `clk_name_gated`, because underscores are legal name characters. The fix is to put braces around the name: `${clk_name}`. The same problem appears when you build a file name such as `$design_final.rpt` and the variable `design_final` does not exist. If a variable lookup fails and the name looks longer than you wrote, check where the name ends.



## 2.5 Order of evaluation inside a command

Command substitution runs from the inside out and from left to right.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [string length [string trim " abc "]]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
3
```

`string trim` runs first, giving `abc`. Then `string length` runs on that result. Finally `puts` prints it. If a command inside brackets fails, the whole outer command fails with the inner error.

## 2.6 Semicolons, newlines and comments

A semicolon ends a command on the same line.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set p 1; set q 2; puts [expr {$p + $q}]
```

This prints `3`.

A comment starts with `#`, but only where a command could start. That is why a comment after a command on the same line needs a semicolon first.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set v 5 ;# fine  
puts $v
```

This prints `5`. Without the semicolon, the `#` would be treated as one more argument to `set`, and the command would fail with a wrong number of arguments.



## 2.7 A command is a list of words

Because a command is split into words, a command can be stored in a variable and run later.

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
set cmd {puts "ok"}
eval $cmd
```

This prints `ok`. The braces kept the command intact until `eval` evaluated it. Later lessons build tool command lines this way, and the next section explains why braces matter there.

## 2.8 Reading tool command lines as words

Let us apply the rule to an Innovus line. To be run at the Innovus prompt. Check the options against your release with `setOptMode -help` and `getOptMode -help`.

**Legacy UI.**

**INNOVUS TCL LEGACY UI** (to be run at the Innovus prompt)

```
setOptMode -opt_add_insts true
set v [getOptMode -opt_add_insts -quiet]
```

The first line has three words: the command `setOptMode`, the option `-opt_add_insts` and the value `true`. Tcl does not know what the option means. It passes the words to the tool, and the tool decides. The second line runs `getOptMode` inside the brackets. With `-quiet`, the command returns only the value, so `v` receives it. Without `-quiet`, the command prints a line to the log and console (name, value, type and whether you set it), which is easier to read but harder to use in a script.



The whole of Tcl syntax is this idea applied again and again: words, three substitutions, braces to prevent them, and quotes to group while still allowing them.

#### 🔗 WHY AN INTERVIEWER ASKS ABOUT THIS

"What is the difference between braces and double quotes?" is one of the most common Tcl questions in CAD interviews. A short answer is that both group words, but quotes allow substitution and braces do not. A better answer adds that Tcl substitutes only once, that braces are used for code bodies and regular expressions, and that `${name}` protects a variable name from the characters that follow it. Expect a follow-up asking you to predict the output of a line like `puts {$x}`.

#### ✓ KEY TAKEAWAYS

- A command is split into words, and three substitutions apply to each word: variable, command and backslash.
- Double quotes group and allow substitution, braces group and prevent it, and substitution happens only once.
- Use `${name}` when characters that could belong to a name follow the variable.
- A comment needs a command start, so put a semicolon before a trailing `#`.



## LESSON 3

# Variables and expressions

Flow scripts keep settings in variables, compute numbers from them and compare results with limits. This lesson covers the commands for that work, the difference between integer and floating-point arithmetic, and one rule about `expr` that avoids a whole class of bugs and speeds up loops.

## 3.1 Setting and changing variables

A variable is created the first time you set it. There are no declarations and no types.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set count 0
incr count
incr count 5
puts $count
set s "clk"
append s "_gated" "_en"
puts $s
```

This prints:

**OUTPUT** *(what the commands above print)*

```
6
clk_gated_en
```

`incr` adds an integer, one by default, and creates nothing if the variable is missing, so set it first. `append` adds text to the end of a variable without copying it, which is the efficient way to build a long string piece by piece.



## 3.2 Does it exist?

Reading a variable that was never set is an error. Check first when a variable is optional.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [info exists nosuch]
set nosuch 1
puts [info exists nosuch]
unset nosuch
puts [info exists nosuch]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
0
1
0
```

`info exists` returns 1 or 0 and never fails. `unset` removes a variable, which is useful when a script must not carry a setting from one step to the next.

## 3.3 Arithmetic with expr

Arithmetic is not part of the language syntax. It is done by the `expr` command, which takes an expression and evaluates it.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [expr {7 / 2}]
puts [expr {7 / 2.0}]
puts [expr {7 % 3}]
puts [expr {2 ** 10}]
puts [expr {int(2.7)}]
puts [expr {round(2.5)}]
puts [expr {sqrt(16)}]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
3
3.5
1
1024
2
3
4.0
```

The first line is the common trap: two integers divide as integers, so `7 / 2` is `3`. Make one of them a floating-point number to get `3.5`. This matters in timing arithmetic. Dividing a period of `2` by `4` gives `0`, while dividing `2.0` by `4` gives `0.5`. Write time values with a decimal point.

Common functions include `abs`, `int`, `round`, `sqrt`, `pow`, `min` and `max`.

## 3.4 Always brace the expression

Compare these four lines.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set a 4
puts [expr $a*2]
puts [expr {$a*2}]
set b "3 + 4"
puts [expr $b]
puts [expr {$b}]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
8
8
7
3 + 4
```

The unbraced form gave the right answer for `a` but treated the contents of `b` as part of the expression, so it evaluated `3 + 4` and returned `7`. The braced form saw `$b` as one value and returned it unchanged. Two reasons to brace every expression follow.



1. **Safety.** An unbraced expression is substituted by Tcl first and then evaluated again by `expr`, so a variable that holds text with special characters can change the meaning, or even run a command.
2. **Speed.** A braced expression can be compiled once and reused. In a loop that runs thousands of times, the difference is large.

Make `expr {...}` a habit.

## 3.5 Comparisons and conditions

Comparison operators are `==`, `!=`, `<`, `>`, `<=`, `>=`. For strings, use `eq` and `ne`. Logical operators are `&&`, `||` and `!`. The conditional operator `? :` picks one of two values.

**TCL EXAMPLE** (type at the `tclsh` prompt, or put in a script)

```
puts [expr {"1.0" == 1}]
puts [expr {"abc" eq "abc"}]
puts [expr {5 > 3 && 2 > 1}]
puts [expr {-0.12 < 0 ? "violated" : "met"}]
```

This prints:

**OUTPUT** (what the commands above print)

```
1
1
1
violated
```

The first result shows that `==` compares numerically when both sides look like numbers, so `"1.0" == 1` is true. If you need exact text equality, use `eq`. A slack value of `-0.12` compared with zero gives a violated setup path, which is the sort of test a flow script makes after a timing step.



## 3.6 Units and conversions

Tcl does not know about nanoseconds or picofarads. The number is just a number, and the unit comes from the tool's library or settings. A script that mixes units gives wrong answers without any error.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set period_ns 2.0
set load_pf 0.020
puts "period in ps: [expr {$period_ns * 1000}]"
puts "load in fF: [expr {$load_pf * 1000}]"
```

This prints:

**OUTPUT** *(what the commands above print)*

```
period in ps: 2000.0
load in fF: 20.0
```

When a tool returns a number, check the unit it uses before comparing it with a limit. Constraint files and reports can use different units, and the unit is stated in the library or in the tool's settings, not in the number.

## 3.7 Floating-point surprises

Floating-point numbers are stored in binary, so some decimal values are not exact. Formatting hides the difference when you print.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [format "%.3f" 0.1]
puts [format "%.2f" 1.005]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
0.100
1.00
```



The second line shows `1.005` rounding to `1.00`, because the stored value is slightly below `1.005`. For timing work this is rarely a problem, but do not test two computed floats for equality. Compare with a small tolerance, for example `expr {abs($a - $b) < 1e-9}`.

## 3.8 A small Innovus example

Let us consider a script that asks for more timing paths than the default.

**Legacy UI.** To be run at the Innovus prompt. Check the options against your release with `timeDesign -help`.

**INNOVUS TCL** **LEGACY UI** *(to be run at the Innovus prompt)*

```
set n 100
timeDesign -postCTS -numPaths $n
```

The variable holds the integer, and Tcl substitutes it into the command. The `timeDesign` command takes an integer for `-numPaths`, with a default of 50 in the Legacy reference. Changing the number in one place, at the top of the script, is the whole value of using a variable here.

### Q WHY AN INTERVIEWER ASKS ABOUT THIS

Expect "why do you put `expr` expressions in braces" and "what does `expr {7/2}` return". The strong answer to the first gives both reasons, safety and speed. For the second, say `3`, explain that integers divide as integers, and show that `7/2.0` returns `3.5`. A follow-up may ask how you would compare two delay numbers that came from different reports, and the expected answer mentions units first.

### ✓ KEY TAKEAWAYS

- `set`, `incr` and `append` create and change variables, and `info exists` checks without failing.
- Integers divide as integers, so write time values with a decimal point.
- Always write `expr {...}` with braces for safety and speed.
- Tcl numbers carry no units, so check the unit of any value a tool returns.



## LESSON 4

# Lists

Cell names, net names, file names and path records are all lists. A script that works well with lists can select, count, sort and loop over almost anything a tool returns. This lesson covers the list commands you will use daily, the nested lists that hold records, and the mistakes that come from treating a list as plain text.

## 4.1 What a list is

A Tcl list is a string with a particular structure: elements separated by spaces, with braces or quotes around an element that itself contains spaces. Every command that works on lists understands this structure.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set cells {u1 u2 u10 ff_a ff_b}  
puts [llength $cells]  
puts [lindex $cells 0]  
puts [lindex $cells end]  
puts [lrange $cells 1 2]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
5  
u1  
ff_b  
u2 u10
```

Indexes start at zero, and `end` means the last element. `end-1` is the one before it. `lrange` returns a new list, and it never fails for indexes outside the list: it returns what exists.



## 4.2 Building and changing lists

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
lappend cells ff_c
puts $cells
set cells [linsert $cells 1 u_new]
puts $cells
set cells [lreplace $cells 1 1]
puts $cells
```

This prints:

**OUTPUT** *(what the commands above print)*

```
u1 u2 u10 ff_a ff_b ff_c
u1 u_new u2 u10 ff_a ff_b ff_c
u1 u2 u10 ff_a ff_b ff_c
```

`lappend` changes the variable in place. `linsert` and `lreplace` return a new list, so assign the result. The `list` command builds a list from arguments, and it quotes elements for you.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set l [list a "b c" d]
puts [llength $l]
puts $l
set s "a b c d"
puts [llength $s]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
3
a {b c} d
4
```

The element `b c` stayed one element, and Tcl showed it with braces. Always create a list with `list` or `lappend`, not by gluing text together, because the commands then handle spaces correctly.



## 4.3 Searching and sorting

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
puts [lsearch $cells ff_a]
puts [lsearch -all -glob $cells ff_*]
puts [lsearch -inline -glob $cells u*]
puts [lsort $cells]
puts [lsort -dictionary $cells]
puts [lsort -decreasing {3 20 100 7}]
puts [lsort -integer -decreasing {3 20 100 7}]
```

This prints:

**OUTPUT** (what the commands above print)

```
3
3 4 5
u1
ff_a ff_b ff_c u1 u10 u2
ff_a ff_b ff_c u1 u2 u10
7 3 20 100
100 20 7 3
```

`lsearch` returns the index of the first match, or `-1`. With `-all` it returns every index, and with `-inline` it returns the matching element itself. The default match is a glob pattern, so `ff_*` works as in a shell. Pass `-exact` if you need an exact comparison.

Sorting needs care. The default sort is alphabetical, so `u10` comes before `u2`. Use `-dictionary` to sort embedded numbers as numbers, and use `-integer` or `-real` for lists of numbers. The line `lsort -decreasing {3 20 100 7}` returned `7 3 20 100`, because as text `7` is the largest. The `-integer` option fixes it.

## 4.4 Looping

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
foreach c {u1 ff_a} { puts "cell $c" }
foreach {name type} {u1 AND2 ff_a DFF} { puts "$name is $type" }
```



This prints:

**OUTPUT** *(what the commands above print)*

```
cell u1
cell ff_a
u1 is AND2
ff_a is DFF
```

`foreach` can take several variables at once, which reads a flat list in pairs, triples and so on. This is how a list of alternating names and values is processed.

## 4.5 Taking lists apart

`lassign` copies elements into variables. Extra elements are returned.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
lassign {ff_a ff_b -0.12} start end slack
puts "$start -> $end : $slack"
set rest [lassign {1 2 3 4} p q]
puts "$p $q rest=$rest"
```

This prints:

**OUTPUT** *(what the commands above print)*

```
ff_a -> ff_b : -0.12
1 2 rest=3 4
```

The expansion operator `{*}` turns a list into separate arguments. It is the clean way to pass a stored list of options to a command.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set args {-from ff_a -to ff_b}
puts [list report_timing {*}$args]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
report_timing -from ff_a -to ff_b
```

Without `{*}`, the list would be one argument. Section 11.4 returns to this.

## 4.6 Transforming a list with lmap

`lmap` is like `foreach` but collects a result for each element.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [lmap c {u1 u2 u3} {string toupper $c}]
puts [lmap n {1 2 3 4 5 6} {expr {$n % 2 ? [continue] : $n * 10}}]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
U1 U2 U3
20 40 60
```

The second line uses `continue` inside the body to skip the odd numbers, so only the even ones are transformed. `lmap` arrived in Tcl 8.6, so check the version of your tool's Tcl before using it in a flow script.

## 4.7 Splitting and joining text

`split` turns text into a list at the characters you give. `join` does the reverse.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [split "top/core/u_alu/add1" /]
puts [join {top core u_alu} /]
puts [split "a,b,,c" ,]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
top core u_alu add1
top/core/u_alu
a b {} c
```

This is how a hierarchical instance name is cut into levels. Notice that two adjacent separators give an empty element, shown as `{}`.

## 4.8 Nested lists as records

A list of lists is a table. Each inner list is one record.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set paths {{ff_a ff_b -0.12} {ff_c ff_d 0.30} {ff_e ff_f -0.45}}
puts [lsort -real -index 2 $paths]
foreach p [lsort -real -index 2 $paths] {
    lassign $p s e sl
    puts [format "%-6s %-6s %7.2f" $s $e $sl]
}
puts [lindex $paths 1 2]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
{ff_e ff_f -0.45} {ff_a ff_b -0.12} {ff_c ff_d 0.30}
ff_e    ff_f      -0.45
ff_a    ff_b      -0.12
ff_c    ff_d       0.30
0.30
```

`lsort -index 2` sorts the records by their third field, and `-real` compares as numbers, so the worst slack comes first. `lindex $paths 1 2` reaches into a nested list: record 1, field 2. A flow script that holds one record per path in a list of lists can sort, filter and print timing results with the commands above.



## 4.9 Pitfalls

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set bad "a {b"
catch {llength $bad} err
puts $err
puts [lrange {a b c d} 1 end-1]
puts [llength [lrange {a b c d} 5 9]]
puts [lindex {a b c} 7]
puts [llength [lindex {a b c} 7]]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
unmatched open brace in list
b c
0
0
```

Three lessons are in that output. A string with an unbalanced brace is not a valid list. `lrange` past the end quietly returns an empty list. And `lindex` with an index that does not exist returns an empty string rather than an error, so a loop that runs one step too far does not stop on its own. Be careful with `<=` against `llength` in a `for` loop.

## 4.10 A small Innovus example

A tool command that returns many names gives you a list to work with. In the Legacy command reference, `dbGet` walks the database by a dotted path and returns a list or a string. Its example `dbGet top.insts.cell.name` returns the master name of every instance. To be run at the Innovus prompt. Check the details against your release with `dbGet -help`.



## Legacy UI.

**INNOVUS TCL** **LEGACY UI** (to be run at the Innovus prompt)

```
set masters [dbGet top.insts.cell.name]
puts "instances: [llength $masters]"
set kinds [lsort -unique $masters]
puts "different masters: [llength $kinds]"
```

The reference notes that `dbGet` returns a list or a string, so every command in this lesson applies to the result. The `-u` option of `dbGet` returns unique results directly, which avoids sorting in Tcl.

### 🔗 WHY AN INTERVIEWER ASKS ABOUT THIS

Expect "how do you loop over a list of cells", "what is the difference between `lappend` and `linsert`" and "why did `lsort` put `u10` before `u2`". Strong answers use `foreach`, mention that `lappend` changes in place while `linsert` returns a copy, and name `-dictionary`, `-integer` and `-real`. A practical question may ask you to sort path records by slack, which is the `lsort -real -index` pattern above.

### ✓ KEY TAKEAWAYS

- A list is a string with a structure, so build lists with `list` and `lappend`, and loop with `foreach`.
- `lsearch` finds, `lsort` orders (choose `-dictionary`, `-integer` or `-real` for the data), and `lassign` and `{*}` take lists apart.
- Nested lists hold records, and `lsort -index` sorts them by a field.
- Out-of-range `lrange` and `lindex` return empty values without an error, so check the bounds yourself.



## LESSON 5

# Strings and regular expressions

Much EDA scripting is reading text that a tool wrote: a log full of messages, a report with a slack number three lines down, a list of names with brackets in them. This lesson covers the `string` command for simple jobs, `format` and `scan` for numbers, and `regexp` and `regsub` for patterns.

## 5.1 The string command

`string` is a family of subcommands. The ones you will use most:

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set s "report_timing -late"
puts [string length $s]
puts [string range $s 0 12]
puts [string first "_" $s]
puts [string last "_" $s]
puts [string index $s 0]
puts [string toupper $s]
puts [string trim " x "]
puts [string trimleft "00042" 0]
puts [string repeat "-" 12]
puts [string map {_ -} $s]
puts [string equal abc abc]
puts [string compare abc abd]
puts [string is integer -strict 42]
puts [string is double -strict abc]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
19
report_timing
6
6
r
REPORT_TIMING -LATE
x
42
-----
report-timing -late
1
-1
1
0
```

`string first` and `string last` return an index, or `-1` if the text is not found. `string map` replaces several substrings in one pass. `string is` tests what kind of text a value is: `integer`, `double`, `alpha`, `digit` and others. Use `-strict` so that an empty string is not reported as a number.

## 5.2 Matching with glob patterns

`string match` compares text against a glob pattern: `*` matches any run of characters, `?` one character, and `[abc]` one of a set.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [string match {ff_*} ff_a]
puts [string match {u[0-9]} u7]
puts [string match -nocase {FF*} ff_a]
```

This prints three lines of `1`. Glob patterns are the same style tools use for names, so they are the natural choice for selecting cells and ports. For anything more structured, use a regular expression.



## 5.3 format and scan

`format` builds text from values, in the style of C's `printf`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [format "%-8s|%6d|%8.3f" cell 42 -0.1234]
puts [format "%05d" 42]
puts [format "%x" 255]
puts [format "%s has %d regs" acc8 9]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
cell    |    42|  -0.123
00042
ff
acc8 has 9 regs
```

`%-8s` pads a string to eight characters on the left, `%6d` pads an integer to six characters, and `%8.3f` prints a float with three decimals in a field of eight. Use these to line up the columns of a summary table.

`scan` is the reverse: it reads values out of text according to a format.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
scan "WNS -0.123 ns" "WNS %f %s" v unit
puts "$v $unit"
```

This prints `-0.123 ns`. `scan` is handy for fixed layouts. For lines whose layout varies, regular expressions are more robust.

## 5.4 Regular expressions with regexp

`regexp` tests whether text matches a pattern and can capture parts of it. The result is `1` for a match and `0` for none. Put the pattern in braces so that Tcl does not process the backslashes first.



Let us consider a log line. The message ids in the following example are ones that the Innovus reference lists, and the line itself is synthetic.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set line "**ERROR: (ENCTS-423): no library found for cell u1"
puts [regexp {\([A-Z]+\)-(\d+)\)} $line whole pkg num]
puts "$whole | $pkg | $num"
puts [regexp {nomatch} $line]
puts [regexp -inline {\([A-Z]+\-\d+)\)} $line]
puts [regexp -inline -all {[0-9]+} "a1 b22 c333"]
puts [regexp -all {l} "hello world"]
puts [regexp -nocase {error} $line]
puts [regexp {^*\*(ERROR|WARN)} $line m sev]
puts $sev
```

This prints:

**OUTPUT** *(what the commands above print)*

```
1
(ENCTS-423) | ENCTS | 423
0
(ENCTS-423) ENCTS-423
1 22 333
3
1
1
1
ERROR
```

The pattern `\([A-Z]+\)-(\d+)\)` looks for a literal bracket, capital letters, a hyphen, digits and a closing bracket. Each pair of parentheses without a backslash is a **capture group**. After the pattern, the variables receive the whole match first and then one value per group. With `-inline`, the matches are returned as a list instead. With `-all`, every match is found, and the return value is the number of matches.

The commonly used pieces:

- `.` matches any character, `*` zero or more, `+` one or more, `?` zero or one.
- `\d` is a digit, `\s` white space, `\w` a word character. `[a-z]` is a range, and `^[a-z]` is anything except a lower-case letter.
- `^` and `$` anchor to the start and end of the text.
- `(a|b)` means either one.



## 5.5 Why the pattern goes in braces

A backslash in double quotes is consumed by Tcl before the regular expression engine sees it.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [regexp "\\d+" "abc123"]
puts [regexp {\d+} "abc123"]
puts [regexp "\d+" "abc123"]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
1
1
0
```

The third form failed, because Tcl turned `\d` in quotes into a plain `d`. Using braces avoids the problem and is easier to read.

## 5.6 Greedy and lazy

By default `*` and `+` match as much as possible. Add `?` after them to match as little as possible.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [regexp -inline {<(.)>} "<a> <b>"]
puts [regexp -inline {<(.*?)>} "<a> <b>"]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
{<a> <b>} {a> <b>}
<a> a
```

The greedy pattern ran to the last closing bracket. The lazy one stopped at the first. When a pattern captures more than you expected, a greedy match is usually the reason.



## 5.7 Substitution with regsub

`regsub` replaces matches. Use `\1`, `\2` for captured groups, and `-all` to replace every match.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [regsub {u(\d+)} "u1 u22" {inst_\1}]
puts [regsub -all {u(\d+)} "u1 u22" {inst_\1}]
set n [regsub -all {\s+} "a  b  c" " " "clean"]
puts "$n: $clean"
```

This prints:

**OUTPUT** *(what the commands above print)*

```
inst_1 u22
inst_1 inst_22
2: a b c
```

Without a variable name, `regsub` returns the new text. With a variable name at the end, it stores the text there and returns the number of replacements.

## 5.8 Names with brackets

Bus bits such as `top/u1[3]` contain square brackets, which are special for Tcl, for glob patterns and for regular expressions. Each of them needs care.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [regsub -all {[^a-z0-9_]} {top/u1[3]} _]
catch {regsub -all {[^a-z0-9_]} "top/u1[3]" _} err
puts $err
```

This prints:

**OUTPUT** *(what the commands above print)*

```
top_u1_3_
invalid command name "3"
```



In the braced form, Tcl left the text alone. In the quoted form, Tcl treated `[3]` as a command to run and failed. The same applies to a name held in a variable: keep it in braces when you write it, and escape the brackets in a glob pattern with a backslash, as in `string match {ff_sum_reg\ [0\]} $name`.

## 5.9 Parsing a value out of a report

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set row "Slack Time    -0.234"
if {[regexp {Slack Time\s+(-?[0-9.]+)} $row -> slack}} {
    puts "slack=$slack"
}
```

This prints `slack=-0.234`. The pattern allows any amount of white space and an optional minus sign. The variable `->` is only a name. Many scripts use it to show that the whole match is being ignored. The skeleton of this example, a `regexp` inside an `if`, is the basis of every report parser.

## 5.10 A small Innovus example

The Legacy `getOptMode` command prints each parameter as a line in the form `-opt_add_insts true # bool, default=true`, followed by the bare value. The `redirect` command can send command output to a Tcl variable with its `-variable` option. To be run at the Innovus prompt. Check the option order with `redirect -help`.

**Legacy UI.**

**INNOVUS TCL LEGACY UI** *(to be run at the Innovus prompt)*

```
redirect optmodes {getOptMode -opt_add_insts} -variable
if {[regexp {(-opt_add_insts)\s+(\S+)\s+#} $optmodes -> name value}} {
    puts "$name is $value"
}
```

The first line captures the printed text in the variable `optmodes`, and the second pulls out the parameter name and value. For a single value, `getOptMode ... -quiet` is simpler, as Lesson 2



showed. Capturing and parsing becomes valuable when a command prints a larger block of text that has no `-quiet` form.

#### 🔗 WHY AN INTERVIEWER ASKS ABOUT THIS

Typical questions: "how do you extract a number from a report line", "what does greedy mean in a regular expression", and "why is the pattern in braces". Answers should show a `regexp` with a capture group, name the lazy `?` modifier, and explain that braces stop Tcl from consuming backslashes. A practical question may hand you a few log lines and ask for a count of errors by message id, and Lesson 9 builds that tool.

#### ✓ KEY TAKEAWAYS

- `string` handles cutting, searching and testing, `string match` uses glob patterns, and `format` and `scan` convert between text and numbers.
- `regexp` tests and captures, and `-inline` and `-all` return values instead of a count.
- Put patterns in braces so that Tcl does not consume the backslashes.
- Names with brackets need braces or escapes everywhere, and `regsub -all` is the tool for cleaning them.



## LESSON 6

# Control flow and error handling

A flow script has to decide: continue if timing is met, stop if a step fails, and treat each file in a directory the same way. This lesson covers the decision and loop commands, and then the part that matters most in long runs, handling errors so a failure is recorded and does not silently ruin the rest of the flow.

## 6.1 if and elseif

The test is an expression, so it uses the rules of `expr`. The condition and the bodies are in braces.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc status {slack} {  
    if {$slack < 0} {  
        return violated  
    } elseif {$slack < 0.05} {  
        return marginal  
    } else {  
        return met  
    }  
}  
puts [status -0.1]  
puts [status 0.02]  
puts [status 0.4]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
violated  
marginal  
met
```



Always put the opening brace on the same line as `if` and `else`. A newline before a brace ends the command, and the next line becomes a separate command that fails.

## 6.2 switch

`switch` compares one value with several patterns.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc kind {name} {  
    switch -glob -- $name {  
        ff_*    { return register }  
        u_*     { return gate }  
        default { return other }  
    }  
}  
puts [kind ff_a]  
puts [kind u_1]  
puts [kind clk]  
switch -exact -- setup { setup - hold { puts "a check" } default { puts "unknown" } }
```

This prints:

**OUTPUT** *(what the commands above print)*

```
register  
gate  
other  
a check
```

`-glob` matches with glob patterns, `-exact` (the default) compares whole strings, and `-regexp` uses regular expressions. The `--` ends the options, so a value that starts with a minus sign is not read as an option. A body written as `-` means "use the body of the next pattern", which is how `setup - hold` shares a body.



## 6.3 Loops

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
set i 0
while {$i < 3} { puts "i=$i"; incr i }
for {set k 0} {$k < 3} {incr k} { puts "k=$k" }
foreach a {1 2 3} b {x y} { puts "$a $b" }
```

This prints:

**OUTPUT** (what the commands above print)

```
i=0
i=1
i=2
k=0
k=1
k=2
1 x
2 y
3
```

`while` repeats while a condition is true. `for` has an initial step, a test and a step after each pass. `foreach` walks one or more lists. When lists have different lengths, the shorter one is padded with empty values, as the last line shows.

`break` leaves the loop, and `continue` skips to the next pass.

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
foreach v {3 -1 5 -2 7} {
    if {$v < 0} { continue }
    if {$v > 6} { break }
    puts "v=$v"
}
```

This prints `v=3` and `v=5`. The negative values were skipped, and the loop stopped at `7`.



## 6.4 Errors

An error stops the current script and passes up to the caller, unless something catches it. The `error` command raises one on purpose.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc need_positive {x} {  
    if {$x <= 0} { error "value must be positive: $x" }  
    return $x  
}
```

## 6.5 catch

`catch` runs a script and returns a code instead of failing: `0` for normal completion, `1` for an error. The result or the error message goes into the variable you name, and an optional second variable receives a dictionary with details.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

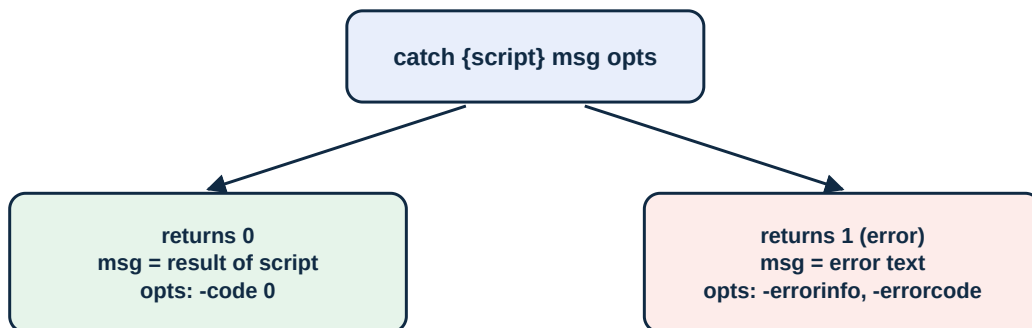
```
set rc [catch {expr {1/0}} msg]  
puts "rc=$rc msg=$msg"  
set rc [catch {set ok 5} msg]  
puts "rc=$rc msg=$msg"  
set rc [catch {error "step failed" "detail"} msg opts]  
puts "rc=$rc msg=$msg"  
puts [dict get $opts -code]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
rc=1 msg=divide by zero  
rc=0 msg=5  
rc=1 msg=step failed  
1
```



**Figure 6.1: What catch returns**

Test the return value, then read msg and opts.  
Return codes 2, 3 and 4 mean return, break and continue.

*catch returns 0 or 1 and stores the result or the error in the variable you name.*

**What to notice.** Test the return value first, then read the message and the options dictionary.

Figure 6.1 shows how the pieces fit together. The usual shape in a flow script is:

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
if {[catch {need_positive -2} msg opts]} {  
    puts "failed: $msg"  
}
```

The `opts` dictionary holds `-code`, `-errorinfo` (a stack trace) and `-errorcode` (a machine-readable list). The stack trace is the most useful when you need to find where a deep error began.

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
catch {need_positive -1} m o  
puts [dict get $o -errorinfo]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
value must be positive: -1
  while executing
"error "value must be positive: $x" "
  (procedure "need_positive" line 2)
  invoked from within
"need_positive -1"
```

## 6.6 return -code and custom codes

A procedure can return an error without calling `error`, by using `return -code error`. The caller sees the same behaviour.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc myerr {} { return -code error "custom failure" }
puts [catch myerr m]
puts $m
```

This prints:

**OUTPUT** *(what the commands above print)*

```
1
custom failure
```

You can also pass a structured `-errorcode` to `error`, so a handler can decide what to do without parsing text.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
catch {error "no lib" "" {EDA NOLIB u1}} m o
puts [dict get $o -errorcode]
```

This prints `EDA NOLIB u1`. The convention is a short list: a family, a kind and some detail. A handler that tests the first two elements is more reliable than one that searches the message.



## 6.7 try, on error and finally

Tcl 8.6 adds `try`, which reads more clearly than `catch` for multiple cases and for clean-up.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
try {
    need_positive -5
} trap {} {err} {
    puts "caught: $err"
} finally {
    puts "cleanup done"
}
set r [try { need_positive 3 } on error {e} { puts bad; set e } ]
puts $r
try { error "boom" } on error {msg info} {
    puts "msg=$msg code=[dict get $info -code]"
}
```

This prints:

**OUTPUT** *(what the commands above print)*

```
caught: value must be positive: -5
cleanup done
3
msg=boom code=1
```

The `finally` body always runs, whether or not an error happened. That is the place to close a file or restore a setting. Because `try` is a Tcl 8.6 command, check your tool's Tcl version before relying on it, and use `catch` where the tool ships an older version.

## 6.8 A loop that consumes a queue

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set queue {a b c}
while {[llength $queue] > 0} {
    set queue [lassign $queue head]
    puts "processing $head"
}
```



This prints `processing a`, `processing b` and `processing c`. `lassign` removes the first element and returns the rest, which is a compact way to work through a to-do list.

## 6.9 A small Innovus example

Let us consider a step that saves the design. If the save fails, the script should record it and stop, rather than carry on. To be run at the Innovus prompt. Check the options against your release with `saveDesign -help`.

### Legacy UI.

**INNOVUS TCL** **LEGACY UI** *(to be run at the Innovus prompt)*

```
if {[catch {saveDesign test.enc} msg opts]} {  
    puts "saveDesign failed: $msg"  
    exit 1  
}
```

The reference for `saveDesign` lists `test.enc -timingGraph` as a usage example, so the file name and the options are ones you can check in your release. The `catch` stops a failing save from being hidden in a long log, and the non-zero `exit` tells a calling shell script or scheduler that the run failed.

#### 🔗 WHY AN INTERVIEWER ASKS ABOUT THIS

Interviewers like "how do you make a flow script stop when a step fails" and "what is the difference between `catch` and `try`". A strong answer says that `catch` returns a code and puts the message and details into variables, that `try` adds `on error`, `trap` and `finally` in 8.6, and that a flow should log the failure and exit with a non-zero status. A common follow-up is "how do you make sure a file is closed even on error", which is the `finally` clause.



**✓ KEY TAKEAWAYS**

- `if`, `switch`, `while`, `for` and `foreach` take their conditions as `expr` rules and their bodies in braces, with the opening brace on the same line.
- `catch` turns an error into a return code, and the dictionary of details holds the stack trace and the error code.
- `try` with `finally` is the clean way to guarantee clean-up in Tcl 8.6.
- Log the failure and exit with a non-zero status, so the people and tools that call your script can see it.



## LESSON 7

# Procedures, scope and namespaces

A script that repeats the same five lines for each stage of a flow is a script waiting for a typing error. A procedure gives those lines a name, takes the parts that change as arguments, and returns a result. This lesson covers how to write procedures, where their variables live, how to pass a variable by name, and how to keep related procedures together.

## 7.1 Defining a procedure

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc half {x} { return [expr {$x / 2.0}] }
puts [half 5]
proc slack_of {required arrival {margin 0.0}} {
    return [expr {$required - $arrival - $margin}]
}
puts [slack_of 2.0 1.75]
puts [slack_of 2.0 1.75 0.125]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
2.5
0.25
0.125
```

`proc` takes a name, a list of arguments and a body. An argument written as `{margin 0.0}` has a default value, so the caller may leave it out. A procedure returns the value of `return`, or the value of its last command if `return` is absent.



If a caller gives the wrong number of arguments, Tcl says so:

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
catch {half} err
puts $err
catch {half 1 2} err
puts $err
```

Both print `wrong # args: should be "half x"`.

## 7.2 A variable number of arguments

An argument called `args` at the end collects all remaining arguments into a list.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc total {args} {
    set sum 0
    foreach a $args { set sum [expr {$sum + $a}] }
    return $sum
}
puts [total]
puts [total 1 2 3]
```

This prints `0` and `6`. This is the pattern used by tool commands that accept any number of cell names or options.



## 7.3 Returning more than one value

Return a list and take it apart with `lassign`, or return a dictionary.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc stats {values} {  
    set n [llength $values]  
    set sum 0.0  
    foreach v $values { set sum [expr {$sum + $v}] }  
    return [list $n [expr {$sum / $n}]]  
}  
lassign [stats {1.0 2.0 4.5}] n mean  
puts "n=$n mean=$mean"
```

This prints `n=3 mean=2.5`. A dictionary is clearer when there are many fields, and Lesson 8 shows how to build and read one.

## 7.4 Where variables live

A procedure has its own local variables. A name used inside a procedure is not the same as a name outside, even if they look the same.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set x global_value  
proc show {} {  
    set x local_value  
    return $x  
}  
puts [show]  
puts $x
```

This prints:

**OUTPUT** *(what the commands above print)*

```
local_value  
global_value
```



To use a variable from the top level inside a procedure, declare it with `global`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc bump {} { global counter; incr counter }  
set counter 0  
bump; bump  
puts $counter
```

This prints `2`. Without the declaration, the procedure cannot see the variable:

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
catch {proc noglobal {} { return $counter }; noglobal} err  
puts $err
```

This prints `can't read "counter": no such variable`.

Globals are convenient for settings that every step reads, such as a design name. Use them sparingly. A procedure that depends on several globals is hard to test, and passing values as arguments is clearer.

## 7.5 Passing a variable by name with upvar

A procedure receives values, not variables. To let it change a variable of the caller, pass the **name** and use `upvar` to make a local alias.

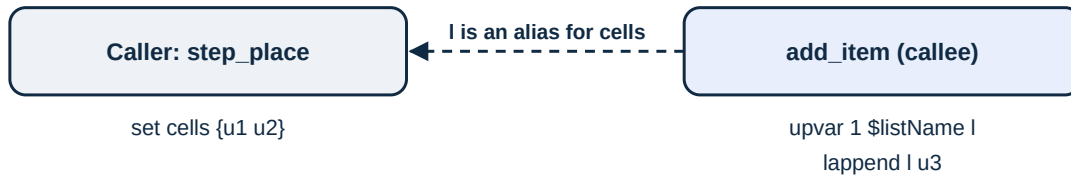
**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc add_item {listName item} {  
    upvar 1 $listName l  
    lappend l $item  
}  
set cells {}  
add_item cells u1  
add_item cells u2  
puts $cells
```



This prints `u1 u2`. Figure 7.1 shows the alias. The `1` means one level up in the call stack, which is the caller. Notice that the call writes `cells`, not `$cells`. With `$cells`, the procedure would receive the contents and could not change the variable.

**Figure 7.1: upvar makes an alias**



`upvar` makes a local name point at a variable in the caller's frame.

**Pass the NAME of the variable, not its value.**

*The callee's name `l` points at the caller's variable `cells`.*

**What to notice.** The call passes the name `cells`, not its value, so `lappend` changes the caller's list.

The same technique writes a swap:

**TCL EXAMPLE** (type at the `tclsh` prompt, or put in a script)

```

proc swap {aName bName} {
    upvar 1 $aName a $bName b
    set t $a; set a $b; set b $t
}
set p 1; set q 2
swap p q
puts "$p $q"
  
```

This prints `2 1`.



## 7.6 Running code in the caller with uplevel

`uplevel` evaluates a script in the caller's context. With it you can write your own control commands.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc repeat {n body} {  
    for {set i 0} {$i < $n} {incr i} { uplevel 1 $body }  
}  
set hits 0  
repeat 3 { incr hits }  
puts $hits
```

This prints `3`. The body ran in the caller, so it could see and change `hits`. This is also how wrappers such as a `timed` procedure in Lesson 11 run the code you pass to them.

## 7.7 Returning a record

A procedure that finds the worst path can return a dictionary with named fields.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc worst {paths} {  
    set best ""  
    foreach p $paths {  
        lassign $p s e sl  
        if {$best eq "" || $sl < [dict get $best slack]} {  
            set best [dict create start $s end $e slack $sl]  
        }  
    }  
    return $best  
}  
set w [worst {{ff_a ff_b -0.12} {ff_e ff_f -0.45}}]  
puts $w  
puts [dict get $w end]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
start ff_e end ff_f slack -0.45
ff_f
```

Each field has a name, so the caller does not need to remember which position holds the slack.

## 7.8 Namespaces

As a script grows, procedure names collide. A **namespace** groups related names under a prefix, with `::` as the separator. Variables inside a namespace live with it and stay out of the global space.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
namespace eval flow {
    variable steps_run 0
    namespace export run_step
    proc run_step {name} {
        variable steps_run
        incr steps_run
        return "ran $name ($steps_run)"
    }
    proc count {} { variable steps_run; return $steps_run }
}
puts [flow::run_step floorplan]
puts [flow::run_step place]
puts [flow::count]
namespace eval util { proc run_step {name} { return "util says $name" } }
puts [util::run_step x]
namespace eval other { namespace import ::flow::run_step }
puts [other::run_step cts]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
ran floorplan (1)
ran place (2)
2
util says x
ran cts (3)
```

Two namespaces can each have a `run_step` without a clash. `variable` inside a procedure connects it to the namespace's variable. `namespace export` lists the names others may import, and `namespace import` makes them available without the prefix. For a flow library, one namespace per area, such as `flow`, `report` and `util`, keeps names predictable.

## 7.9 Looking inside

`info` answers questions about procedures, which helps when you read a colleague's script.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [info args slack_of]
puts [string trim [info body half]]
puts [info commands half]
puts [info procs slack*]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
required arrival margin
return [expr {$x / 2.0}]
half
slack_of
```

## 7.10 A small Innovus example

A procedure that picks a timing stage turns a name into the tool's option. To be run at the Innovus prompt. Check the options against your release with `timeDesign -help`.



## Legacy UI.

INNOVUS TCL **LEGACY UI** (to be run at the Innovus prompt)

```
proc time_stage {stage {paths 50}} {  
    switch -exact -- $stage {  
        preCTS    { timeDesign -preCTS -numPaths $paths }  
        postCTS   { timeDesign -postCTS -numPaths $paths }  
        postRoute { timeDesign -postRoute -numPaths $paths }  
        default   { error "unknown stage: $stage" }  
    }  
}  
time_stage postCTS 100
```

The reference for `timeDesign` lists `-preCTS`, `-postCTS` and `-postRoute` as stage selectors and `-numPaths` with a default of 50, so the procedure uses the same default. The procedure rejects an unknown stage with a clear message, so a typo is caught on the first line of the log.

### 🔍 WHY AN INTERVIEWER ASKS ABOUT THIS

Typical questions: "what does `upvar` do", "how do you pass an array or list to a procedure and change it", and "when would you use a namespace". The strong answers say that `upvar` aliases a caller variable by name, that you pass the name and not the value, and that namespaces prevent clashes and keep state tidy. A follow-up may ask for the difference between `global` and `variable`, which differ in where the variable lives: the top level or the current namespace.

### ✓ KEY TAKEAWAYS

- A procedure takes arguments, can have defaults and an `args` tail, and returns a value or a record.
- Variables are local unless you use `global`, `variable` or `upvar`.
- To change the caller's variable, pass its name and use `upvar 1`.
- Namespaces keep procedure names and state apart, and `info` lets you inspect them.



## LESSON 8

# Arrays and dictionaries

A list is good when you have a sequence. A flow script also needs to answer "what is the value for this name": the slack of each view, the number of warnings for each message id, the settings for each stage. Tcl has two keyed containers for this. The **array** is a variable with named elements. The **dictionary** is a value that holds key and value pairs, so it can be passed to a procedure and returned from one.

## 8.1 Arrays

An array element is written `name(key)`. There is no declaration.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set slack(setup) -0.12
set slack(hold) 0.04
puts $slack(setup)
puts [array exists slack]
puts [lsort [array names slack]]
puts [array size slack]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
-0.12
1
hold setup
2
```



`array names` returns the keys, in no guaranteed order, so sort them when the order matters. `array get` returns a flat list of keys and values, and `array set` loads one. `parray` prints a whole array in a readable form.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
array set limits {max_fanout 16 max_tran 0.30}  
parray limits
```

This prints:

**OUTPUT** *(what the commands above print)*

```
limits(max_fanout) = 16  
limits(max_tran)  = 0.30
```

An array is not a value. You cannot write `puts $slack` or pass `$slack` to a procedure. To give a procedure access, pass the array name and use `upvar`, as in Lesson 7, or use `array get`. This is the main reason many scripts prefer dictionaries.

## 8.2 Dictionaries

A dictionary is a list of keys and values with a set of commands that treat it as a map.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set d [dict create effort high period 2.0]  
dict set d margin 0.1  
puts [dict get $d period]  
puts [dict exists $d margin]  
puts [dict keys $d]  
dict unset d margin  
puts [dict size $d]
```



The result is `2.0`, then `1`, then `effort period margin`, then `2`. A dictionary keeps its keys in the order they were added, which makes printed output predictable. Reading a key that does not exist is an error, so test with `dict exists` first when the key is optional.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
if {[dict exists $d margin]} { puts "has margin" } else { puts "no margin" }
```

This prints `no margin`, because the key was removed. Tcl 8.7 adds `dict getdef` to read a key with a default, but Tcl 8.6, which most tools ship, does not have it. Use `dict exists` as shown.

Walking a dictionary uses `dict for`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set views [dict create view1 -0.12 view2 0.30 view3 0.02]
dict for {name sl} $views {
    puts [format "%-6s %6.2f" $name $sl]
}
puts [dict keys $views]
dict unset views view2
puts [dict keys $views]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
view1  -0.12
view2   0.30
view3   0.02
view1 view2 view3
view1 view3
```



## 8.3 Counting with dict incr

The commonest use is a tally. `dict incr` creates the key at zero if it is missing.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set counts {}
foreach id {ENCTS-423 IMPOPT-1 ENCTS-423 IMPOPT-1 ENCTS-423} {
    dict incr counts $id
}
puts $counts
```

This prints `ENCTS-423 3 IMPOPT-1 2`. Lesson 9 uses this to count the messages in a log. The ids here are placeholders for the pattern. A second command, `dict lappend`, collects values per key.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set groups {}
dict lappend groups warn w1
dict lappend groups err e1
dict lappend groups warn w2
puts $groups
```

This prints `warn {w1 w2} err e1`.

## 8.4 Merging and updating

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set base {effort high period 2.0}
set over {period 2.5}
puts [dict merge $base $over]
dict update base period p effort e {
    set p 2.5
}
puts [dict get $base period]
```

`dict merge` gives a new dictionary in which later values replace earlier ones, which is how a default settings dictionary is combined with a user override. `dict update` links dictionary keys to



local variables for the length of a body, and writes the changes back. This prints `effort high period 2.5` and then `2.5`.

## 8.5 Sorting by value

A dictionary has no sort command. Convert it to a list of pairs, sort, and print.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set area [dict create u_a 12 u_b 3 u_c 40]
set pairs {}
dict for {k v} $area { lappend pairs [list $k $v] }
foreach p [lsort -integer -decreasing -index 1 $pairs] {
    lassign $p k v
    puts [format "%-4s %3d" $k $v]
}
```

This prints:

**OUTPUT** *(what the commands above print)*

```
u_c    40
u_a    12
u_b     3
```

The pattern, collect pairs and `lsort -index 1`, is reused in every "top N" report.

## 8.6 Nested dictionaries

A value can itself be a dictionary, which gives a small tree. Use the multi-key forms to reach inside.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set t {}
dict set t func max setup -0.12
dict set t func min hold 0.04
puts $t
puts [dict get $t func max setup]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
func {max {setup -0.12} min {hold 0.04}}  
-0.12
```

The keys are `func`, `max`, `setup`, and the value is `-0.12`. A structure of this shape is a natural fit for results by view, by corner and by check type.

## 8.7 Transforming and filtering

Tcl 8.6 has `dict map` and `dict filter`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [dict map {k v} {a 1 b 2 c 3} {expr {$v*10}}]  
puts [dict filter {a 1 b 2 c 3} script {k v} {expr {$v>1}}]
```

This prints `a 10 b 20 c 30` and then `b 2 c 3`. The first scales every value. The second keeps only the pairs whose value is greater than one.

## 8.8 Array or dictionary?

Use a dictionary when you will pass the data to a procedure, return it, or print it. Use an array when many procedures update the same table in place, or when you want the per-element trace and `parray`. In a new flow library, dictionaries are the usual choice because they behave like any other value.

## 8.9 A small Innovus example

Let us consider a per-view timing table. The Legacy reference lists `timeDesign` with `-postRoute`, `-expandedViews` and `-outDir`, and it writes one report per view. To be run at the Innovus prompt. Check the options against your release with `timeDesign -help`.



## Legacy UI.

**INNOVUS TCL** **LEGACY UI** *(to be run at the Innovus prompt)*

```
timeDesign -postRoute -expandedViews -outDir sample
```

The step produces one report for each analysis view. A script then reads the worst slack from each report and keeps them in one dictionary, keyed by view name:

**TCL** **EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set wns [dict create]
dict set wns func_ss_setup -0.12
dict set wns func_ff_hold 0.04
dict for {view value} $wns {
    puts [format "%-14s %7.3f %s" $view $value [expr {$value < 0 ? "VIOLATED" :
"met"}]]
}
```

The two numbers here are made up to show the table shape. In a real script they come from parsing the reports, which Lesson 12 shows in a pure-Tcl script. The dictionary keeps the view name and the number together, so a later step can sort the views, count the failing ones or write the table to a file.

### 🔗 WHY AN INTERVIEWER ASKS ABOUT THIS

Typical questions: "what is the difference between an array and a dictionary", "how do you count how many times each message appears" and "how do you pass an array to a procedure". The strong answers say that a dictionary is a value and an array is a variable, show `dict incr` for the count, and explain that an array is passed by name with `upvar` or converted with `array get`. A follow-up may ask what happens when a key is missing, and the answer is an error, so use `dict exists`.



**✓ KEY TAKEAWAYS**

- An array is a variable with named elements, and `array names` gives the keys in no fixed order.
- A dictionary is a value, so it can be passed, returned and printed, and it keeps keys in insertion order.
- `dict incr` and `dict lappend` build tallies and groups in one line each.
- Sort a dictionary by converting it to a list of pairs, and use `dict exists` before reading an optional key.



## LESSON 9

# Files, directories and external programs

Every EDA run leaves files behind: logs, timing reports, netlists, checkpoints. A script that can open them, find them, write new ones and call other programs can automate the work between tool runs. This lesson covers those commands and ends with a small tool that summarises a log by message id.

## 9.1 Writing a file

`open` returns a **channel** name. You write with `puts`, and you must `close` the channel when done.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set fh [open notes.txt w]
puts $fh "line one"
puts $fh "line two"
close $fh
```

The second argument is the mode: `w` creates or overwrites, `a` appends, `r` reads. Forgetting `close` can leave the last lines unwritten, so pair every `open` with a `close`. Lesson 6 showed that `try ... finally` guarantees the close even if an error occurs.



## 9.2 Reading a file

There are two styles. `read` takes the whole file into one string. `gets` reads one line at a time.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set fh [open notes.txt r]
set data [read $fh]
close $fh
puts [string length $data]
puts [llength [split [string trimright $data] \n]]

set fh [open notes.txt r]
while {[gets $fh line] >= 0} { puts "got: $line" }
close $fh
```

This prints:

**OUTPUT** *(what the commands above print)*

```
18
2
got: line one
got: line two
```

`gets` returns the length of the line, or `-1` at the end of the file, so `>= 0` is the correct loop test. An empty line gives `0`, which is why the test is not simply "true". For a large log, read line by line. A multi-gigabyte file in one `read` can exhaust memory.

When a file cannot be opened, `open` raises an error. Catch it to give a clear message:

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [catch {open nosuch.txt r} err]
puts $err
```

This prints `1` and then `couldn't open "nosuch.txt": no such file or directory`.



## 9.3 The file command

`file` is a family of subcommands for names and the file system.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set fh [open notes.txt a]; puts $fh "line three"; close $fh
puts [file exists notes.txt]
puts [file size notes.txt]
puts [file extension notes.txt]
puts [file rootname notes.txt]
puts [file tail /a/b/c.rpt]
puts [file dirname /a/b/c.rpt]
puts [file join /a b c.rpt]
file mkdir out/rpt
puts [file isdirectory out/rpt]
file copy notes.txt out/rpt/notes.bak
```

This prints:

**OUTPUT** *(what the commands above print)*

```
1
29
.txt
notes
c.rpt
/a/b
/a/b/c.rpt
1
```

Build paths with `file join` rather than gluing strings with slashes, and take names apart with `file tail`, `file dirname`, `file rootname` and `file extension`. `file mkdir` creates all missing directories in the path. Other subcommands include `file rename`, `file delete`, `file mtime` and `file normalize`.



## 9.4 Finding files with glob

`glob` returns the names that match a pattern, in the same glob style as `string match`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [glob -directory out/rpt *]
foreach n {a b} { set f [open $n.tarpt w]; puts $f x; close $f }
puts [lsort [glob *.tarpt]]
puts [catch {glob *.zzz} err]
puts $err
puts [glob -nocomplain *.zzz]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
out/rpt/notes.bak
a.tarpt b.tarpt
1
no files matched glob pattern "*.zzz"
```

`glob` raises an error when nothing matches, and the last line is empty because `-nocomplain` returns an empty list instead. Use `-nocomplain` when "no files" is a normal case, and sort the result, because the order is not defined.

## 9.5 Running other programs with exec

`exec` runs a program and returns what it wrote to standard output.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [exec echo hello]
puts [catch {exec false} err opts]
puts $err
puts [lrange [dict get $opts -errorcode] 0 0]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
hello
1
child process exited abnormally
CHILDSTATUS
```

Two rules. First, a program that exits with a non-zero status makes `exec` raise an error, so wrap it in `catch`. The `-errorcode` holds `CHILDSTATUS`, the process id and the exit status, which tells the caller which. Second, any text a program writes to standard error also turns into an error unless you redirect it, so `exec cmd 2>@1` or `exec cmd 2>/dev/null` is common.

## 9.6 Environment and arguments

Environment variables are in the array `env`. A script started from a shell receives its arguments in `argv`, with their number in `argc`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [info exists env(HOME)]
```

This prints `1`. A script that takes a file name checks `argc` first:

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
if {$argc != 1} { puts stderr "usage: count_messages.tcl logfile"; exit 2 }
set logfile [lindex $argv 0]
```

Run as `tclsh count_messages.tcl` with no argument, it prints the usage line and exits with status 2. The same variables are available when a script is run from a tool's batch mode, which Lesson 11 covers.



## 9.7 Counting messages in a log

Let us consider a log in which each message line starts with a severity and a message id in brackets. The file below is synthetic, written to show the pattern. The message ids come from the Innovus reference, but the lines are not real tool output.

**FILE** sample.log *(synthetic, not real tool output)*

```
Innovus synthetic log (not real tool output)
**WARN: (ENCTS-3): clock net skew group not found, using default
**ERROR: (ENCTS-423): no library found for cell u1
**WARN: (NRIG-1323): net n_42 has no route guide
**ERROR: (ENCTS-423): no library found for cell u7
**WARN: (CHKCTS-16): sink pin ff_a/CK has no clock arrival
**WARN: (ENCTS-3): clock net skew group not found, using default
Finished step place
**WARN: (NRIG-1323): net n_77 has no route guide
**WARN: (ENCTS-3): clock net skew group not found, using default
```

The tool combines three ideas from earlier lessons: a `regexp` with capture groups (Lesson 5), `dict incr` for a tally (Lesson 8), and a sort of pairs.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
if {$argc != 1} { puts stderr "usage: count_messages.tcl logfile"; exit 2 }
set fh [open [lindex $argv 0] r]
set counts [dict create]
set sev_total [dict create]
while {[gets $fh line] >= 0} {
    if {[regexp {^*\*(ERROR|WARN): \(([A-Z]+-[0-9]+)\)} $line -> sev id]} {
        dict incr counts "$sev $id"
        dict incr sev_total $sev
    }
}
close $fh
set rows {}
dict for {k v} $counts { lappend rows [list $k $v] }
foreach r [lsort -integer -decreasing -index 1 $rows] {
    lassign $r k v
    puts [format "%-18s %3d" $k $v]
}
dict for {sev n} $sev_total { puts "total $sev: $n" }
```



Run with `tclsh count_messages.tcl sample.log`, it prints:

**OUTPUT** *(what the commands above print)*

```
WARN ENCTS-3          3
ERROR ENCTS-423       2
WARN NRIG-1323        2
WARN CHKCTS-16        1
total WARN: 6
total ERROR: 2
```

The lines that do not start with a severity are ignored, so the header and "Finished step place" cost nothing. The most frequent warning comes first, which is the one worth looking at before anything else. This is the shape of a log reviewer. The exercise in Appendix B asks you to extend it.

## 9.8 A small Innovus example

Innovus can send command output to a file with `redirect`, and `source` runs a script file. The reference lists `redirect timing.txt "timeDesign -postRoute" -append -tee` as a pattern, and `source` takes `-quiet`, `-verbose` or `-echo`. To be run at the Innovus prompt. Check the options with `redirect -help` and `source -help`.

### Legacy UI.

**INNOVUS TCL** **LEGACY UI** *(to be run at the Innovus prompt)*

```
redirect timing.txt "timeDesign -postRoute" -append -tee
source -echo my_flow.tcl
```

The first line writes the output of the timing step both to the screen and to the file `timing.txt`. The second runs a script and echoes each command to the log, which makes a long flow readable afterwards.

The timing step also writes its reports to a directory. The reference states that `timeDesign` writes per-group reports with the extension `.tarpt` into `./timingReports` by default, and that



`-outDir` and `-prefix` change the directory and file-name prefix. A script can then collect them with `glob`:

**INNOVUS TCL** **LEGACY UI** *(to be run at the Innovus prompt)*

```
foreach rpt [lsort [glob -nocomplain timingReports/*.tarpt]] {  
    puts "report: [file tail $rpt]"  
}
```

The `-nocomplain` keeps the script quiet if the directory is empty, and `file tail` strips the directory from the name.

#### 🔍 WHY AN INTERVIEWER ASKS ABOUT THIS

Typical questions: "how do you read a file line by line", "how do you run a shell command from Tcl and get its status" and "write a script that counts the errors in a log". Good answers use `gets` in a while loop, `catch {exec ...}` with `-errorcode`, and `regexp` plus `dict incr`. A follow-up may ask what happens if the file does not exist, and the answer is that `open` raises an error, which you catch.

#### ✓ KEY TAKEAWAYS

- Always pair `open` with `close`, and read large files with `gets` in a loop that tests for `>= 0`.
- Use `file join`, `file tail` and the other `file` subcommands rather than string tricks for paths.
- `glob` fails on no match unless you give `-nocomplain`, and its order is not defined, so sort it.
- `exec` raises an error on a non-zero exit status or any text on standard error, so wrap it in `catch`.



## LESSON 10

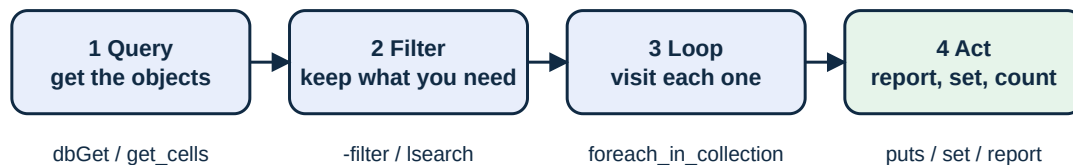
# Working with design objects

Most useful scripts in a physical design flow do four things: ask the tool for a set of objects, narrow the set, go through it, and act on each item. The objects are cells, nets, pins, ports, clocks. This lesson practises that pattern on plain Tcl lists, so you can run it anywhere, and then shows the commands Innovus provides to do the same on a real design.

## 10.1 Query, filter, loop, act

Figure 10.1 shows the four steps. The first step is the only one that talks to the tool. After it, you hold a set of names or objects, and the rest is ordinary Tcl.

**Figure 10.1: Query, filter, loop, act**



*Ask for as little as you can, then loop over the small result.*

***Only the first step needs the tool.***

**What to notice.** Ask the tool for as little as you can, then loop over the small result.



Let us consider a small design written as a list of records, each holding an instance name and its master (library cell) name.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set insts {
  {u_alu/reg_a DFFX1}
  {ff_sum_reg[0] DFFX1}
  {ff_sum_reg[1] DFFX1}
  {u_alu/g1 AND2X1}
  {u_ctl/g2 AND2X1}
  {u_ctl/inv1 INVX1}
}
```

This stands in for the answer to a query. The records are invented for the lesson.

## 10.2 Filter

Select the instances whose name contains `reg`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set hits {}
foreach i $insts {
  if {[string match *reg* [lindex $i 0]]} { lappend hits [lindex $i 0] }
}
puts $hits
```

This prints `u_alu/reg_a {ff_sum_reg[0]} {ff_sum_reg[1]}`. Tcl shows the names with brackets in braces, which is how it protects them inside a list.



## 10.3 Group by block

The block is the first part of the hierarchical name. Names with no slash belong to the top level.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set blocks [dict create]
foreach i $insts {
    lassign $i name master
    set parts [split $name /]
    set blk [expr {[llength $parts] > 1 ? [lindex $parts 0] : "(top)"}]
    dict incr blocks $blk
}
dict for {b n} $blocks { puts "$b: $n" }
```

This prints:

**OUTPUT** *(what the commands above print)*

```
u_alu: 2
(top): 2
u_ctl: 2
```

## 10.4 A histogram of masters and the total area

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set area [dict create DFFX1 12.6 AND2X1 5.4 INVX1 3.6]
set hist [dict create]
set tot 0.0
foreach i $insts {
    dict incr hist [lindex $i 1]
    set tot [expr {$tot + [dict get $area [lindex $i 1]]}]
}
foreach m [lsort [dict keys $hist]] { puts "$m [dict get $hist $m]" }
puts "total area [format %.1f $tot]"
```



This prints:

**OUTPUT** *(what the commands above print)*

```
AND2X1 2
DFFX1 3
INVX1 1
total area 52.2
```

The areas are invented numbers to show the arithmetic. In a real flow, the tool reports the area, and you should check which unit the library uses.

## 10.5 Names with brackets

Bus bits are the usual trouble. In a glob pattern, `[0]` means "the character 0", so it will not match a name that contains the literal text `[0]`.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [string match {ff_sum_reg\[0\]} {ff_sum_reg[0]}]
puts [string match {ff_sum_reg[0]} {ff_sum_reg[0]}]
puts [regsub -all {[[]]} {ff_sum_reg[0]} _]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
1
0
ff_sum_reg_0_
```

The first form escapes the brackets and matches. The second reads `[0]` as a character set and fails. The `regsub` replaces both bracket characters, which is useful when a name becomes part of a file name. Lesson 5 showed why the quoted form of such a name breaks, and it applies here too: keep names with brackets in braces, or build them with `list`.

## 10.6 Collections and lists

Innovus returns many objects in a **collection**, which is a handle to a set of objects that lives inside the tool. A collection is not a Tcl list. You cannot call `lindex` on it, and printing it shows a



handle rather than names. Three commands work on collections: `sizeof_collection` counts them, `sort_collection` orders them by a property, and `foreach_in_collection` walks them. To get a string out of one object, use `get_property`.

Other commands, such as `dbGet`, return an ordinary Tcl list or string. The reference states this, so every list command in Lessons 4 to 8 applies. When you read a script, check which kind you hold.

## 10.7 Innovus: collections

To be run at the Innovus prompt. Check the details against your release with the `-help` option of each command.

### Legacy UI.

```
INNOVUS TCL  LEGACY UI  (to be run at the Innovus prompt)

puts [sizeof_collection [all_registers -flops]]
foreach_in_collection c [get_cells *] {
    puts [get_property $c hierarchical_name]
}
set sorted [sort_collection [all_inputs] hierarchical_name]
```

The reference documents the first as the flop-count idiom, the second as the pattern for printing every cell name, and the third as the sorting example. Each pass of `foreach_in_collection` holds a one-object collection, which is why `get_property` is needed to read its name.

The `get_cells` command accepts a `-filter` expression on any cell property. The reference lists properties such as `area`, `ref_lib_cell_name` and `hierarchical_name`, and shows this pattern:

```
INNOVUS TCL  LEGACY UI  (to be run at the Innovus prompt)

get_cells "o*" -filter "@ref_lib_cell_name != FD2 && @is_black_box==false"
```

In this form, `@` introduces a property name. Filtering inside the tool is usually faster than fetching everything and filtering in Tcl.



## 10.8 Innovus: database queries

`dbGet` walks the database by a dotted path and returns a Tcl list or string.

### Legacy UI.

**INNOVUS TCL** **LEGACY UI** *(to be run at the Innovus prompt)*

```
set masters [dbGet top.insts.cell.name]
set n [dbGet top.numInsts]
set names [dbGet top.terms.name]
```

The reference lists `top.insts.cell.name` for master names, `top.numInsts` for the instance count and `top.terms.name` for the top ports. Adding `-u` returns unique results. A pattern after the path filters: `dbGet top.insts.cell.name INV1` selects instances of that master, and `-p2` goes back two levels to return the instance objects instead. Use `dbGet -help` and the `??` suffix on a path to list the attributes of an object in your release. The reference also warns never to paste the pointers it prints into other commands or into another session.

A change is made with `dbSet`, which sets an attribute that `dbGet` shows as settable. The reference gives `dbSet top.insts.pStatus fixed` as a pattern. This changes every instance, so read it twice before running it on a design you care about.

## 10.9 Innovus: Common UI

The Common UI describes the same data as attributes. The only Common UI forms in the lab kit used for this course are the following. They are shown as the lab writes them, and the lab text says they are used with the synthesis flow.

### Common UI.

**INNOVUS TCL** **COMMON UI** *(to be run at the Innovus prompt)*

```
get_db scan_segments *spi*
set_db inst:dtmf_recvr_core/PM_INST/isolation_enable_reg_reg .dft_dont_scan true
```

`get_db` reads an object or attribute, and `set_db` writes an attribute. The first line, from the scan-chain lab, returns the scan segments whose names contain `spi`. The second keeps one



register out of scan. Do not mix Legacy and Common UI commands in a single script, and for anything beyond these forms, check the Common UI guide for your release.

#### 🔍 WHY AN INTERVIEWER ASKS ABOUT THIS

Typical questions: "how do you get all the flops in a design and count them", "what is the difference between a collection and a list" and "how do you match a net name that has brackets". A strong answer uses `all_registers -flops` with `sizeof_collection`, says that a collection is a handle inside the tool and must be walked with `foreach_in_collection`, and escapes the brackets in the pattern. Interviewers also like the question "how would you find the cells of one master in a hierarchical block", where the expected reply is a `get_cells -filter` or a `dbGet` pattern.

#### ✓ KEY TAKEAWAYS

- The four steps are query, filter, loop and act, and only the first needs the tool.
- A collection is a handle inside the tool, so use `sizeof_collection`, `sort_collection`, `foreach_in_collection` and `get_property` on it.
- `dbGet` returns an ordinary Tcl list or string, so all the list and dictionary commands apply.
- Escape brackets in glob patterns, keep names with brackets in braces, and filter inside the tool when it is faster.



## LESSON 11

# Parameterised flow scripts

A script that works for one design on one day is a recording. A flow script is a program: the design name, the clock period and the effort level are inputs, each step reports whether it worked, and a failed step does not leave you guessing. This lesson builds the pieces: a settings file, commands built from data, a few small library commands, command-line arguments, a step runner and checkpoints.

## 11.1 Keep settings in one file

Put the values that change from run to run in a file of their own, and load it with `source`.

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
# config.tcl
set period 2.0
set effort medium
```

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
source config.tcl
puts "$period $effort"
```

This prints `2.0 medium`. Anyone who wants a different run edits the config file and not the flow. For several designs, keep one config file per design and pass its name as an argument, as Section 11.6 shows.



## 11.2 Building a command from data

Sometimes the command to run is assembled by the script. The safe way is to build it as a list, because `list` quotes each element.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set cmd [list report_timing -from ff_a -to "u b"]
puts $cmd
puts [llength $cmd]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
report_timing -from ff_a -to {u b}
5
```

The name `u b` has a space and stays one argument, which it would not if you had glued the text together with string operations. A list built this way can be run with `eval $cmd`, and in Tcl 8.5 and later the brace expansion in Section 11.4 is the neater way.

`subst` is the second tool. It performs variable and command substitution on a string that you have kept in braces, which makes a template.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set tpl {period=$period effort=$effort}
puts [subst $tpl]
```

This prints `period=2.0 effort=medium`. Templates suit report headers and file names.

## 11.3 Evaluating

`eval` joins its arguments into one script and runs it. It is rarely needed now, and it has a danger: if the pieces are not lists, a space inside a value splits it. Prefer `{*}` or a procedure. When you do use `eval`, pass a list that was built with `list`.



## 11.4 Brace expansion: `{*}`

The expansion operator `{*}` turns a list into separate arguments. It is the right way to pass stored options to a command.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set opts {-from ff_a -to ff_b}
puts [llength [list report_timing {*} $opts]]
puts [llength [list report_timing $opts]]
```

This prints `5` and then `2`. With `{*}` the four options are separate words. Without it, the whole list is one argument, and a command that expects `-from` as its own word would reject it. The same operator is how a flow script keeps one list of common options and adds it to several commands. Lesson 4 introduced it, and this is where it earns its place.

## 11.5 Small library commands

A few commands appear in almost every flow script.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts [clock format 0 -gmt 1 -format {%Y-%m-%d %H:%M:%S}]
puts [package require Tcl 8.5]
puts [catch {package require nosuchpkg} err]
puts $err
```

This prints:

**OUTPUT** *(what the commands above print)*

```
1970-01-01 00:00:00
8.6.12
1
can't find package nosuchpkg
```

`clock format` turns a time in seconds into text, and `clock seconds` gives the current time, which names a log file or stamps a report. The example uses time zero with `-gmt 1` so the output does not change between runs. `package require Tcl 8.5` returns the version of the



interpreter, here 8.6.12 in the test run, and raises an error if it is older than the version you ask for. That makes it a good first line for a script that uses `lmap` or `try`.

## 11.6 Taking arguments

When a script is started with `tclsh script.tcl arg1 arg2`, the arguments are in `argv`, their number is in `argc`, and the script name is in `argv0`. A script that checks them can be reused.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts $argc
puts $argv
puts $argv0
```

Run as `tclsh argtest.tcl in.rpt 0.1`, it prints `2`, `in.rpt 0.1` and `argtest.tcl`. Check `argc` first and print a usage line, as Lesson 9 did. Whether a tool sets `argv` for a script depends on how it was started, so print the variable once in your environment before depending on it. For a design tool, the more usual route is to set the variables in a small wrapper file that sources the flow.



## 11.7 A step runner

A flow is a list of steps. A runner executes each one, catches errors, and prints a table at the end. A failing step is recorded and the flow decides what to do next.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc run_step {name body} {
    if {[catch {uplevel 1 $body} msg]} { return [list $name FAIL $msg] }
    return [list $name OK $msg]
}

set steps {
    {step_a {string cat a { done}}}
    {step_b {error "library missing"}}
}

set failed 0
foreach s $steps {
    lassign $s name body
    set r [run_step $name $body]
    puts [format "%-8s %-5s %s" {*}$r]
    if {[lindex $r 1] eq "FAIL"} { incr failed }
}

puts "failed=$failed"
```

This prints:

**OUTPUT** *(what the commands above print)*

```
step_a  OK    a done
step_b  FAIL  library missing
failed=1
```

The body runs in the caller's scope with `uplevel`, so steps can use the flow's variables. The runner combines `catch` from Lesson 6, `lassign` from Lesson 4, `format` from Lesson 5 and `{*}` from this lesson. At the end, a non-zero `exit` when `failed` is above zero tells the shell script or scheduler that the flow did not finish cleanly. Appendix B has an exercise that extends it.



## 11.8 A timing wrapper

A wrapper takes a script as an argument, runs it in the caller, and reports around it.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc timed {label body} {
    set t0 [clock milliseconds]
    set rc [catch {uplevel 1 $body} res]
    set dt [expr {[clock milliseconds] - $t0}]
    puts "$label done"
    return -code $rc $res
}
set x 0
timed loop { for {set i 0} {$i < 1000} {incr i} { incr x } }
puts $x
```

This prints `loop done` and `1000`. The elapsed time is in `dt`. A real version would print it, but the number differs on every run, so the example leaves it out. `return -code $rc $res` passes on whatever the body did, including an error, so the wrapper does not hide failures.

## 11.9 A small Innovus example

A flow can save a checkpoint after each major step so a failed step does not mean a restart from the beginning. The reference states that `saveDesign` writes a restore script with the name you give plus a data directory, and that you source that script later, or use `restoreDesign`. To be run at the Innovus prompt. Check the options with `saveDesign -help` and `restoreDesign -help`.

**Legacy UI.**

**INNOVUS TCL LEGACY UI** *(to be run at the Innovus prompt)*

```
source -echo config.tcl
redirect flow.log "timeDesign -postCTS" -append -tee
if {[catch {saveDesign after_cts.enc} msg]} {
    puts "checkpoint failed: $msg"
    exit 1
}
```

The first line echoes the config file into the log, which records the settings used. The `redirect` pattern comes from the Legacy reference and sends the timing step to the screen and to a log.



The `catch` around `saveDesign` is the one from Lesson 6. In a later run, `source after_cts.enc` or `restoreDesign` brings the saved state back, and the options are listed in the reference. Keep a checkpoint before every large step.

#### ❓ WHY AN INTERVIEWER ASKS ABOUT THIS

Typical questions: "how do you make a flow script reusable across designs", "what does `{*}` do" and "how do you keep going when one step fails". A strong answer describes a config file loaded by `source`, arguments from `argv`, `{*}` for option lists, and a step runner with `catch` that records failures and exits non-zero at the end. A follow-up may ask what `eval` does and why `list` is safer than string gluing, and the answer is that `list` quotes each element.

#### ✓ KEY TAKEAWAYS

- Keep run-to-run settings in a config file and `source` it, or pass its name as an argument.
- Build commands as lists, expand stored options with `{*}`, and use `subst` for templates.
- `package require Tcl 8.5` at the top documents the version a script needs.
- A step runner with `catch` and `uplevel` records failures, and the flow ends with a non-zero `exit` when any step failed.



## LESSON 12

# Debugging, speed and a complete script

Scripts fail. A good engineer finds the fault quickly, and the habits for that are the same in every tool. This lesson covers the commands that show what a script is doing, the mistakes that appear again and again, how to measure speed, and finally a complete script that uses nearly everything from the first eleven lessons.

## 12.1 Print first

The cheapest debugger is `puts`. Print the value of a variable before the line that fails, and use braces or `list` so that spaces and empty values are visible.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
puts "name=<$name> n=[length $items]"
```

The angle brackets show whether a value is empty or has trailing spaces. When a list looks wrong, `puts [length $x]` and `puts [lindex $x 0]` usually explain it.

## 12.2 Asking Tcl about itself

`info` answers questions about procedures and variables.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc f {a {b 1} args} { return [list $a $b $args] }  
puts [info args f]  
puts [info default f b v]  
puts $v  
puts [info level]  
puts [info exists wns]
```



This prints:

**OUTPUT** *(what the commands above print)*

```
a b args
1
1
0
0
```

`info args f` lists the parameter names. `info default f b v` returns 1 when the parameter has a default and stores the default in `v`. `info level` is the depth of the call stack: 0 at the top level, and 1 inside a procedure called from there. `info exists` tests a variable, and `info commands` and `info procs` list commands by pattern.

## 12.3 Stack traces

When a procedure fails deep in a chain of calls, the error trace shows the whole path.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc a {} { b }
proc b {} { c }
proc c {} { error "deep failure" }
catch a m o
puts [dict get $o -errorinfo]
```

This prints:

**OUTPUT** *(what the commands above print)*

```
deep failure
  while executing
"error "deep failure" "
  (procedure "c" line 1)
  invoked from within
"c "
  (procedure "b" line 1)
  invoked from within
"b "
  (procedure "a" line 1)
  invoked from within
"a"
```



Read it from the top: the message, where it was raised, and then each caller. In a tool, `errorInfo` holds the same trace for the last error, so print it when a script stops with a one-line message that is not enough.

Error codes carry the family of the failure in a form a script can test:

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
catch {expr {1/0}} m o
puts [dict get $o -errorcode]
catch {open /nonexistent/x r} m o
puts [lindex [dict get $o -errorcode] 0]
```

This prints `ARITH DIVZERO {divide by zero}` and `POSIX`. A handler that checks the first word can respond differently to a missing file and to bad arithmetic.

## 12.4 Watching a variable with trace

`trace add variable` runs a command whenever a variable is read, written or removed. It finds the line that changes a value unexpectedly.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
trace add variable wns write {apply {{n1 n2 op} { upvar #0 $n1 v; puts "$n1 is now $v" }}}
set wns -0.5
set wns 0.1
```

This prints:

**OUTPUT** *(what the commands above print)*

```
wns is now -0.5
wns is now 0.1
```

Remove the trace with `trace remove variable` when you have found the cause, because every write pays for the callback.



## 12.5 Wrapping a command with rename

`rename` gives a command a new name, so you can put a wrapper in its place that logs the call and then calls the original.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
proc g {x} { return [expr {$x * 2}] }
rename g g_orig
proc g {x} { puts "calling g $x"; return [g_orig $x] }
puts [g 4]
```

This prints `calling g 4` and then `8`. The technique logs a tool command without editing every place that calls it. Use it for a short debugging session and take it out afterwards.

## 12.6 Measuring speed

`time` runs a script a number of times and reports the average.

**TCL EXAMPLE** *(type at the tclsh prompt, or put in a script)*

```
set r [time {set s 0; for {set i 0} {$i < 1000} {incr i} {incr s}} 10]
puts $r
```

The result has the form `N microseconds per iteration`. The number depends on the machine, so the example does not print one. Use `time` to compare two ways of doing the same job, for example a braced and an unbraced `expr` from Lesson 3. For a whole tool step, use the stage timer that the tool provides. The Innovus reference lists `time_info` with a stage name, `-current` and `-report` for runtime and memory per stage.

Three habits keep scripts fast. Brace every `expr`. Use `lappend` and `append` rather than rebuilding a long value each pass. Filter inside the tool, as Lesson 10 showed, rather than fetching everything and filtering in Tcl.



## 12.7 Common mistakes

These are the faults seen most often. Each has appeared in an earlier lesson.

**TCL EXAMPLE** (type at the tclsh prompt, or put in a script)

```
set x 3
puts [if {$x == "3.0"} {set r same} else {set r diff}]
puts [string equal $x 3.0]
set l {a b c}
set out {}
for {set i 0} {$i <= [llength $l]} {incr i} { lappend out [lindex $l $i] }
puts [llength $out]
puts "<$out>"
```

This prints:

**OUTPUT** (what the commands above print)

```
same
0
4
<a b c {}>
```

The first pair shows that `==` compares numbers, so `3` and `"3.0"` are equal, and `string equal` compares text, so they differ. Decide which you mean. The loop used `<=` against the length, ran one step too far, and `lindex` returned an empty element instead of failing.

The remaining mistakes in one list:

- **A brace on its own line.** Writing `if {...}` and putting `{` on the next line ends the command early.
- **Unbraced expr.** It is slower and can be unsafe.
- **Integer division.** `7/2` is 3, so write `7/2.0`.
- **A comment with a brace in it** inside a braced body can unbalance the braces.
- **A list built by gluing strings.** Use `list` and `lappend`.
- **Brackets in names.** Brace them and escape them in glob patterns.
- **Forgetting close.** Pair every `open` with a `close`, ideally with `try ... finally`.
- **A global used by surprise.** Pass values as arguments.



## 12.8 A complete script

The script below reads a timing summary, prints a table sorted by slack, and reports the worst slack, the total negative slack and the count of violating paths. It exits with status 1 when any path is below a margin. The report format is synthetic and made up for this course. It is not a real tool format, so a real report needs a different `regex`, written after you have looked at the file.

The input, `sample_timing.rpt`:

**FILE** `sample_timing.rpt` *(synthetic, not real tool output)*

```
# synthetic timing summary (not a real tool format)
view: func_ss_setup  check: setup
path 1  start: ff_a/CK  end: ff_b/D  slack: -0.120
path 2  start: ff_c/CK  end: ff_d/D  slack: 0.300
path 3  start: ff_e/CK  end: ff_f/D  slack: -0.450
path 4  start: ff_g/CK  end: ff_h/D  slack: 0.012
path 5  start: ff_i/CK  end: ff_j/D  slack: 1.250
```



The script, `parse_report.tcl`:

**TCL SCRIPT** (save as a `.tcl` file and run it with `tclsh`)

```
# usage: tclsh parse_report.tcl report_file ?margin?
package require Tcl 8.5

proc parse_paths {file} {
    set fh [open $file r]
    set paths {}
    try {
        while {[gets $fh line] >= 0} {
            if {[regexp {\^path\s+\d+\s+start:\s+(\S+)\s+end:\s+(\S+)\s+slack:\s+
(-?[0-9.]+)} \
                $line -> s e sl]} {
                lappend paths [list $s $e $sl]
            }
        }
    } finally {
        close $fh
    }
    return $paths
}

proc summarise {paths margin} {
    set wns ""
    set tns 0.0
    set nviol 0
    foreach p $paths {
        set sl [lindex $p 2]
        if {$wns eq "" || $sl < $wns} { set wns $sl }
        if {$sl < $margin} {
            set tns [expr {$tns + $sl}]
            incr nviol
        }
    }
    return [dict create wns $wns tns $tns violating $nviol total [llength $paths]]
}

if {$argc < 1 || $argc > 2} {
    puts stderr "usage: parse_report.tcl report_file ?margin?"
    exit 2
}

set file [lindex $argv 0]
set margin [expr {$argc == 2 ? [lindex $argv 1] : 0.0}]
if {[catch {parse_paths $file} paths]} {
    puts stderr "cannot read $file: $paths"
    exit 2
}
```



```

}
foreach p [lsort -real -index 2 $paths] {
    lassign $p s e sl
    puts [format "%-8s %-8s %7.3f  %s" $s $e $sl [expr {$sl < $margin ? "VIOLATED"
: "met"}]]
}
set r [summarise $paths $margin]
puts [format "WNS %.3f  TNS %.3f  violating %d of %d" \
    [dict get $r wns] [dict get $r tns] [dict get $r violating] [dict get $r
total]]
exit [expr {[dict get $r violating] > 0 ? 1 : 0}]

```

Run as `tclsh parse_report.tcl sample_timing.rpt`, it prints:

#### OUTPUT *(what the commands above print)*

```

ff_e/CK  ff_f/D    -0.450  VIOLATED
ff_a/CK  ff_b/D    -0.120  VIOLATED
ff_g/CK  ff_h/D     0.012  met
ff_c/CK  ff_d/D     0.300  met
ff_i/CK  ff_j/D     1.250  met
WNS -0.450  TNS -0.570  violating 2 of 5

```

The exit status is 1, because two paths are below zero. With a margin of `-0.2` the path at `-0.120` counts as met, and only one path violates. With a missing file the script prints `cannot read nofile` and exits 2, and with no argument it prints the usage line and exits 2.

Count the lessons in it. The `regexp` with capture groups is Lesson 5. The `while` with `gets` and the `try ... finally` that closes the file are Lessons 9 and 6. The list of records and `lsort -real -index` are Lesson 4. The dictionary return is Lessons 7 and 8. The `argc` check, the `catch` around the parse and the exit status are Lessons 9, 6 and 11. Nothing here is specific to one tool.

## 12.9 How it fits an Innovus run

In a real flow, the timing step is Innovus, and this script sits after it. The Legacy reference states that `timeDesign` writes its reports under `./timingReports` by default, with `.tarpt` files for path groups, and that `-outDir` and `-prefix` change the location and the file names. A wrapper script runs the step, finds the report files with `glob`, and calls a parser on each.



**Legacy UI.** To be run at the Innovus prompt. Check the options with `timeDesign -help`.

**INNOVUS TCL** **LEGACY UI** (to be run at the Innovus prompt)

```
timeDesign -postCTS -outDir timingReports
foreach rpt [lsort [glob -nocomplain timingReports/*.tarpt]] {
    puts "found [file tail $rpt]"
}
```

Open one real report first and write the `regexp` for the lines you see. The pattern in `parse_report.tcl` is a template. The reference also describes a `.summary` file with WNS, TNS and the violating-path counts per group, which may already hold the numbers you want. The shape of the flow does not change: run the step, find the file, parse it, decide and exit.

#### ❓ WHY AN INTERVIEWER ASKS ABOUT THIS

Typical questions: "how do you debug a Tcl script", "how do you find where a variable changes" and "write a script that reads a report and prints the worst slack". Good answers mention `puts` with brackets around values, `errorInfo` for the trace, `trace add variable` for a watch, and `time` for speed. For the last, they sketch a `regexp` inside a `gets` loop, collect the numbers, and print the minimum, as `parse_report.tcl` does. Ending with a non-zero exit when the check fails shows you think about the flow around the script.

#### ✓ KEY TAKEAWAYS

- Print values with markers, use `info` to inspect procedures, and read `-errorinfo` from the top.
- `trace add variable` finds the line that changes a value, and `rename` wraps a command for logging.
- Measure with `time`, brace every `expr`, and filter inside the tool.
- A complete tool is small: parse with `regexp`, hold records in a list, return a dictionary, and exit with a status the flow can use.



---

# One-page summary

---

## Purpose

This guide teaches the Tcl that EDA tools speak, from the first command to a complete flow script. Every example that is not tool-specific was run in tclsh 8.6, and the Innovus examples are labelled Legacy or Common UI.

## Workflow map

Words and substitution → variables and expressions → lists → strings and regular expressions → control flow and errors → procedures → dictionaries → files → design objects → parameterised flows → debugging.

## Key concepts

- Every line is a command, which is a list of words. Substitution runs once, left to right. Braces switch it off.
- Always brace `expr`. Integers divide as integers, so write time values with a decimal point.
- A list is a string with structure. Build with `list` and `lappend`, sort with the right option, and expand with `{*}`.
- Put regular expressions in braces. Names with brackets need braces and escapes.
- `catch` turns an error into a code. `try ... finally` guarantees clean-up.
- Pass a variable by name with `upvar 1`. Namespaces keep names and state apart.
- A dictionary is a value and an array is a variable. `dict incr` tallies in one line.
- Query, filter, loop, act. A collection is a handle inside the tool, so use the collection commands on it.
- A flow script reads a config file, takes arguments, records each step and exits non-zero on failure.

## Common mistakes

- An unbraced `expr`, or `7/2` when `7/2.0` was meant.
- An opening brace on its own line after `if` or `else`.
- A regular expression in double quotes, so the backslashes vanish.
- Brackets in a name left unbraced or unescaped.
- Forgetting `close`, or a `for` loop that uses `<=` against `llength`.
- Mixing Legacy and Common UI commands in one script.

## Readiness questions

- ☐ Can you say what each substitution does and in what order?
- ☐ Can you count messages in a log by id with a dictionary?
- ☐ Can you pass a list to a procedure and change it?
- ☐ Can you explain a collection versus a list?
- ☐ Does your flow stop, log and exit non-zero when a step fails?
- ☐ Have you checked every Innovus option with `-help` in your release?

## Evidence to keep

The script, its config file, the log it wrote, and the exit status of the run.



## APPENDIX A

# Command cheat sheet

The Tcl commands used in the lessons, grouped by task, followed by the Innovus idioms. Time and unit conventions come from your library and tool settings, not from Tcl.

## Language rules

| Rule                   | Detail                                                                                            |
|------------------------|---------------------------------------------------------------------------------------------------|
| Command shape          | <code>command arg arg ...</code> , first word is the command, a newline or <code>;</code> ends it |
| <code>\$var</code>     | substitute a variable                                                                             |
| <code>[cmd]</code>     | substitute the result of a command                                                                |
| <code>\x</code>        | backslash escape                                                                                  |
| <code>"..."</code>     | grouping, substitutions happen                                                                    |
| <code>{...}</code>     | grouping, nothing happens                                                                         |
| <code>{*}\$list</code> | expand a list into separate arguments                                                             |
| <code>#</code>         | a comment, only where a command may start                                                         |

## Variables and expressions

| Command                    | Use                                     |
|----------------------------|-----------------------------------------|
| <code>set v x</code>       | create or change                        |
| <code>incr v ?n?</code>    | add an integer                          |
| <code>append v s</code>    | add text                                |
| <code>unset v</code>       | remove                                  |
| <code>info exists v</code> | test without failing                    |
| <code>expr {...}</code>    | arithmetic, always braced               |
| <code>eq ne</code>         | string compare inside <code>expr</code> |

## Lists

| Command                  | Use          |
|--------------------------|--------------|
| <code>list a b c</code>  | build        |
| <code>lappend l x</code> | add in place |



| Command                                                                                                            | Use                   |
|--------------------------------------------------------------------------------------------------------------------|-----------------------|
| <code>llength</code> <code>lindex</code> <code>lrange</code>                                                       | measure and pick      |
| <code>linsert</code> <code>lreplace</code>                                                                         | return a changed copy |
| <code>lsearch</code> <code>?-all?</code> <code>?-glob?</code> <code>?-inline?</code>                               | find                  |
| <code>lsort</code> <code>?-dictionary?</code> <code>?-integer?</code> <code>?-real?</code> <code>?-index n?</code> | order                 |
| <code>lassign</code> <code>l a b</code>                                                                            | take apart            |
| <code>lmap</code> <code>v l {body}</code>                                                                          | transform             |
| <code>split</code> <code>s ch</code> <code>join</code> <code>l ch</code>                                           | text and list         |
| <code>foreach</code> <code>v l {body}</code>                                                                       | loop                  |

## Strings

| Command                                                                                                            | Use               |
|--------------------------------------------------------------------------------------------------------------------|-------------------|
| <code>string</code> <code>length</code> <code>range</code> <code>index</code> <code>first</code> <code>last</code> | measure and cut   |
| <code>string</code> <code>match</code> <code>pat s</code>                                                          | glob match        |
| <code>string</code> <code>map</code> <code>{a b} s</code>                                                          | replace           |
| <code>string</code> <code>trim</code> <code>toupper</code> <code>tolower</code> <code>repeat</code>                | tidy              |
| <code>string</code> <code>equal</code> <code>compare</code> <code>is</code>                                        | compare and test  |
| <code>format</code> <code>fmt args</code>                                                                          | printf style      |
| <code>scan</code> <code>s fmt var...</code>                                                                        | read values       |
| <code>regexp</code> <code>pat s ?v...?</code>                                                                      | match and capture |
| <code>regsub</code> <code>?-all?</code> <code>pat s rep ?var?</code>                                               | replace           |

## Control and errors

| Command                                                                                                                                       | Use                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------|----------------------|
| <code>if</code> <code>{c}</code> <code>{..}</code> <code>elseif</code> <code>{c}</code> <code>{..}</code> <code>else</code> <code>{..}</code> | decide               |
| <code>switch</code> <code>?-glob?</code> <code>-- v {pat body ...}</code>                                                                     | choose               |
| <code>while</code> <code>for</code> <code>foreach</code>                                                                                      | loop                 |
| <code>break</code> <code>continue</code>                                                                                                      | leave or skip        |
| <code>error</code> <code>msg</code>                                                                                                           | raise                |
| <code>catch</code> <code>{script}</code> <code>msg</code> <code>?opts?</code>                                                                 | trap, returns 0 or 1 |
| <code>try</code> <code>{..}</code> <code>on error</code> <code>{m o}</code> <code>{..}</code> <code>finally</code> <code>{..}</code>          | trap and clean up    |
| <code>return</code> <code>-code</code> <code>error</code> <code>msg</code>                                                                    | return an error      |



## Procedures and scope

| Command                                          | Use                      |
|--------------------------------------------------|--------------------------|
| <code>proc name {a {b 1} args} {body}</code>     | define                   |
| <code>global v</code>                            | use a top-level variable |
| <code>upvar 1 name local</code>                  | alias a caller variable  |
| <code>uplevel 1 script</code>                    | run in the caller        |
| <code>namespace eval ns {..}</code>              | group names              |
| <code>variable v</code>                          | namespace variable       |
| <code>info args body commands procs level</code> | inspect                  |

## Arrays and dictionaries

| Command                                                                                           | Use                   |
|---------------------------------------------------------------------------------------------------|-----------------------|
| <code>set a(k) v</code> <code>array names a</code> <code>array get a</code> <code>parray a</code> | arrays                |
| <code>dict create</code> <code>dict get</code> <code>dict set</code> <code>dict exists</code>     | basic dictionary      |
| <code>dict keys</code> <code>dict size</code> <code>dict for {k v} d {..}</code>                  | walk                  |
| <code>dict incr</code> <code>dict lappend</code>                                                  | tally and group       |
| <code>dict merge</code> <code>dict update</code> <code>dict map</code> <code>dict filter</code>   | combine and transform |

## Files and programs

| Command                                                                                                                                                             | Use                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------|
| <code>open f r w a</code> <code>close</code>                                                                                                                        | channels                  |
| <code>gets ch var</code> <code>read ch</code> <code>puts ch s</code>                                                                                                | read and write            |
| <code>file exists</code> <code>join</code> <code>tail</code> <code>dirname</code> <code>rootname</code> <code>extension</code> <code>mkdir</code> <code>copy</code> | names and files           |
| <code>glob ?-nocomplain? pattern</code>                                                                                                                             | find files                |
| <code>exec prog args</code>                                                                                                                                         | run a program             |
| <code>\$argc</code> <code>\$argv</code> <code>\$argv0</code> <code>\$env(NAME)</code>                                                                               | arguments and environment |
| <code>exit n</code>                                                                                                                                                 | end with a status         |

## Flow and debugging

| Command                              | Use                |
|--------------------------------------|--------------------|
| <code>source file</code>             | run a script       |
| <code>subst</code> <code>eval</code> | build and run text |



| Command                                              | Use               |
|------------------------------------------------------|-------------------|
| <code>package require Tcl 8.5</code>                 | check the version |
| <code>clock seconds</code> <code>clock format</code> | time              |
| <code>trace add variable v write cmd</code>          | watch             |
| <code>rename old new</code>                          | wrap              |
| <code>time {script} n</code>                         | measure           |

## Innovus Tcl idioms

Examples are from the Innovus command reference and the synthesis lab kit. They are labelled Legacy or Common UI. None was run for this course. Check each option with `-help` in your release. Do not mix the two UIs in one script.

### Legacy UI

| Goal                         | Idiom                                                                                                        |
|------------------------------|--------------------------------------------------------------------------------------------------------------|
| Count flops                  | <code>sizeof_collection [all_registers -flops]</code>                                                        |
| Print every cell name        | <code>foreach_in_collection c [get_cells *] { puts [get_property \$c hierarchical_name] }</code>             |
| Sort by property             | <code>sort_collection [all_inputs] hierarchical_name</code>                                                  |
| Filter cells in the tool     | <code>get_cells "o*" -filter "@ref_lib_cell_name != FD2 &amp;&amp; @is_black_box==false"</code>              |
| Instance count               | <code>dbGet top.numInsts</code>                                                                              |
| Master of every instance     | <code>dbGet top.insts.cell.name</code> (add <code>-u</code> for unique)                                      |
| Top ports                    | <code>dbGet top.terms.name</code>                                                                            |
| Fix all instances            | <code>dbSet top.insts.pStatus fixed</code> (check first)                                                     |
| Timing at a stage            | <code>timeDesign -postCTS -numPaths 100</code>                                                               |
| Per-view reports             | <code>timeDesign -postRoute -expandedViews -outDir sample</code>                                             |
| Log a step                   | <code>redirect timing.txt "timeDesign -postRoute" -append -tee</code>                                        |
| Capture output in a variable | <code>redirect name {getOptMode -opt_add_insts} -variable</code>                                             |
| Run a script, echo commands  | <code>source -echo my_flow.tcl</code>                                                                        |
| Checkpoint                   | <code>saveDesign after_cts.enc</code> , then <code>source after_cts.enc</code> or <code>restoreDesign</code> |
| Read one mode value          | <code>set v [getOptMode -opt_add_insts -quiet]</code> (Lesson 2)                                             |



---

## Common UI (lab kit only)

| Goal               | Idiom                                                                                            |
|--------------------|--------------------------------------------------------------------------------------------------|
| Read scan segments | <code>get_db scan_segments *spi*</code>                                                          |
| Set an attribute   | <code>set_db inst:&lt;name&gt; .dft_dont_scan true</code>                                        |
| Batch start        | <code>innovus -synthesis -stylus -no_gui -files run.tcl -log &lt;name&gt; -<br/>overwrite</code> |



## APPENDIX B

# Exercise kit

Four small scripts, two input files and four expected outputs. Put them in one folder and run each with `tclsh`. Every expected output came from a real run of the script in `tclsh` 8.6.

**⚠ WATCH OUT**

The two input files are synthetic, written for this course. They are not real tool output or a real tool format. The message ids in the log are ones the Innovus reference lists, but the lines are not real Innovus messages. A real report needs its own `regex`, written after you have looked at the file.

## B.1 The scripts

**SCRIPT 1** `count_messages.tcl`

|                     |                                                                       |
|---------------------|-----------------------------------------------------------------------|
| <b>Lesson</b>       | 9                                                                     |
| <b>What it does</b> | Counts the messages in a log by severity and id, most frequent first. |
| <b>Reads</b>        | <code>sample.log</code>                                               |
| <b>Run</b>          | <code>tclsh count_messages.tcl sample.log</code>                      |
| <b>Expected</b>     | <code>count_messages.expected</code> , shown in Lesson 9.             |



**SCRIPT 2** `parse_report.tcl`

|                     |                                                                                                                |
|---------------------|----------------------------------------------------------------------------------------------------------------|
| <b>Lesson</b>       | 12                                                                                                             |
| <b>What it does</b> | Parses a timing summary, prints a table sorted by slack, and gives WNS, TNS and the number of violating paths. |
| <b>Reads</b>        | <code>sample_timing.rpt</code> , and an optional margin                                                        |
| <b>Run</b>          | <code>tclsh parse_report.tcl sample_timing.rpt</code>                                                          |
| <b>Exit status</b>  | 1 if any path is below the margin, 2 for a usage error.                                                        |
| <b>Expected</b>     | <code>parse_report.expected</code> , shown in Lesson 12.                                                       |

**SCRIPT 3** `step_runner.tcl`

|                     |                                                                          |
|---------------------|--------------------------------------------------------------------------|
| <b>Lesson</b>       | 11                                                                       |
| <b>What it does</b> | Runs four steps, records OK or FAIL for each, and exits 1 if any failed. |
| <b>Reads</b>        | Nothing.                                                                 |
| <b>Run</b>          | <code>tclsh step_runner.tcl</code>                                       |
| <b>Exit status</b>  | 1, because one step is built to fail.                                    |



**FILE** step\_runner.tcl

```

# usage: tclsh step_runner.tcl
# Runs a list of steps, records OK or FAIL for each, prints a table,
# and exits 1 if any step failed.
package require Tcl 8.5

proc run_step {name body} {
    if {[catch {uplevel 1 $body} msg]} { return [list $name FAIL $msg] }
    return [list $name OK $msg]
}

set steps {
    {read_config {set period 2.0}}
    {check_period {expr {$period > 0 ? "period ok" : [error "bad period"]}}}
    {load_library {error "library missing"}}
    {report {string cat "report written"}}
}

set failed 0
foreach s $steps {
    lassign $s name body
    set r [run_step $name $body]
    puts [format "%-14s %-5s %s" {*}$r]
    if {[lindex $r 1] eq "FAIL"} { incr failed }
}
puts "failed=$failed"
exit [expr {$failed > 0 ? 1 : 0}]

```

**OUTPUT** (step\_runner.expected)

```

read_config    OK      2.0
check_period   OK      period ok
load_library    FAIL    library missing
report         OK      report written
failed=1

```

**SCRIPT 4** view\_table.tcl

|                     |                                                                                                                |
|---------------------|----------------------------------------------------------------------------------------------------------------|
| <b>Lesson</b>       | 8                                                                                                              |
| <b>What it does</b> | Holds the worst setup and hold slack per view in a nested dictionary and prints a table sorted by setup slack. |
| <b>Reads</b>        | Nothing. The numbers are invented to show the table shape.                                                     |
| <b>Run</b>          | <code>tclsh view_table.tcl</code>                                                                              |



**FILE** view\_table.tcl

```

# usage: tclsh view_table.tcl
# Holds worst setup and hold slack per view in a dictionary,
# prints a table sorted by setup slack, and counts failing views.
package require Tcl 8.5

set wns [dict create]
dict set wns func_ss setup -0.120
dict set wns func_ss hold 0.045
dict set wns func_ff setup 0.310
dict set wns func_ff hold -0.015
dict set wns scan_ss setup 0.020
dict set wns scan_ss hold 0.080

set rows {}
dict for {view checks} $wns {
    lappend rows [list $view [dict get $checks setup] [dict get $checks hold]]
}
set nfail 0
puts [format "%-8s %8s %8s %s" view setup hold status]
foreach r [lsort -real -index 1 $rows] {
    lassign $r view su ho
    set status [expr {($su < 0 || $ho < 0) ? "FAIL" : "pass"}]
    if {$status eq "FAIL"} { incr nfail }
    puts [format "%-8s %8.3f %8.3f %s" $view $su $ho $status]
}
puts "failing views: $nfail of [dict size $wns]"

```

**OUTPUT** (view\_table.expected)

```

view      setup      hold  status
func_ss   -0.120    0.045  FAIL
scan_ss   0.020     0.080  pass
func_ff   0.310    -0.015  FAIL
failing views: 2 of 3

```



## B.2 The input files

**FILE** sample.log *synthetic*

```
Innovus synthetic log (not real tool output)
**WARN: (ENCTS-3): clock net skew group not found, using default
**ERROR: (ENCTS-423): no library found for cell u1
**WARN: (NRIG-1323): net n_42 has no route guide
**ERROR: (ENCTS-423): no library found for cell u7
**WARN: (CHKCTS-16): sink pin ff_a/CK has no clock arrival
**WARN: (ENCTS-3): clock net skew group not found, using default
Finished step place
**WARN: (NRIG-1323): net n_77 has no route guide
**WARN: (ENCTS-3): clock net skew group not found, using default
```

**FILE** sample\_timing.rpt *synthetic*

```
# synthetic timing summary (not a real tool format)
view: func_ss_setup check: setup
path 1 start: ff_a/CK end: ff_b/D slack: -0.120
path 2 start: ff_c/CK end: ff_d/D slack: 0.300
path 3 start: ff_e/CK end: ff_f/D slack: -0.450
path 4 start: ff_g/CK end: ff_h/D slack: 0.012
path 5 start: ff_i/CK end: ff_j/D slack: 1.250
```



## B.3 Practice tasks

### ✍ EXERCISES

1. **Lesson 9.** Change `count_messages.tcl` to take a second argument, `ERROR` or `WARN`, and print only that severity.
2. **Lesson 9.** Record the line number where each id first appears, and print it as a third column.
3. **Lesson 11.** Add a `timed` wrapper to `step_runner.tcl` so each step also prints its time in milliseconds.
4. **Lesson 8.** Add an overall row to `view_table.tcl` with the worst setup slack and the worst hold slack across all views.
5. **Lesson 8.** Write `proc dict_get_default {d key default}` using `dict exists`. Tcl 8.6 has no `dict getdef`.
6. **Lesson 12.** Change `parse_report.tcl` so a line with a malformed slack is counted and reported, and does not stop the run.

